Adaptive Real-Time Scheduling and Resource Management on Multicore Architectures

vom
Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades eines

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von
Stefan Andreas Schorr
geboren in Neunkirchen (Saar)

D 386

Eingereicht am:	14.01.2015
Tag der mündlichen Prüfung:	25.02.2015
Dekan des Fachbereichs:	Prof. Dr.-Ing. Hans D. Schotten
Promotionskomission	
Vorsitzender:	Prof. Dr.-Ing. Steven Liu
Berichterstattende:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
	Prof. Johan Eker

Erklärung gem. § 6 Abs. 3 Promotionsordnung

Ich versichere, dass ich diese Dissertation selbst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die aus den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Dissertation wurde weder als Ganzes noch in Teilen als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht. Es wurde weder diese noch eine andere Abhandlung bei einem anderen Fachbereich oder einer anderen Universität als Dissertation eingereicht.

(Ort, Datum) (Stefan Schorr)

Abstract

Real-time systems are systems that have to react correctly to stimuli from the environment within given timing constraints. Today, real-time systems are employed everywhere in industry, not only in safety-critical systems but also in, e.g., communication, entertainment, and multimedia systems.

With the advent of multicore platforms, new challenges on the efficient exploitation of real-time systems have arisen: First, there is the need for effective *scheduling algorithms* that feature low overheads to improve the use of the computational resources of real-time systems. The goal of these algorithms is to ensure timely execution of tasks, i.e., to provide runtime guarantees. Additionally, many systems require their scheduling algorithm to flexibly react to unforeseen events.
Second, the inherent parallelism of multicore systems leads to contention for shared hardware resources and complicates system analysis. At any time, multiple applications run with varying resource requirements and compete for the scarce resources of the system. As a result, there is a need for an adaptive *resource management*. Achieving and implementing an effective and efficient resource management is a challenging task. The main goal of resource management is to guarantee a minimum resource availability to real-time applications. A further goal is to fulfill global optimization objectives, e.g., maximization of the global system performance, or the user perceived quality of service.

In this thesis, we derive methods based on the *slot shifting* algorithm [1]. Slot shifting provides flexible scheduling of time-constrained applications and can react to unforeseen events in time-triggered systems. For this reason, we aim at designing slot shifting based algorithms targeted for multicore systems to tackle the aforementioned challenges.

The main contribution of this thesis is to present two global slot shifting algorithms targeted for multicore systems. Additionally, we extend slot shifting algorithms to improve their runtime behavior, or to handle non-preemptive firm aperiodic tasks. In a variety of experiments, the effectiveness and efficiency of the algorithms are evaluated and confirmed.

Finally, the thesis presents an implementation of a slot-shifting-based logic into a resource management framework for multicore systems. Thus, the thesis closes the circle and successfully bridges the gap between real-time scheduling theory and real-world implementations. We prove applicability of the slot shifting algorithm to effectively and efficiently perform adaptive resource management on multicore systems.

TO MY PARENTS HEDWIG AND ANDREAS SCHORR.

"And all was well."

Preface

After the five years of my Ph.D. studies here at the Chair of Real-Time Systems in Kaiserslautern, I can truly say: It has been a marvelous adventure offering many intriguing and eye-opening experiences that massively contributed to my personal development. I had the pleasure to enjoy the international environment at the Chair and the opportunity to meet and work with so many different people from all around the world. I consider it an essential experience that I would not like to have missed.

Directly at the beginning, I could participate in a European research project with many partners from industry and universities from different places all around Europe. Subsequently, I had the chance to frequently travel to project meetings and even international conferences in many different countries in the world. I always enjoyed traveling and I am very pleased that I could see so much of the world in such a short time during the last years. It has been exciting to meet so many fascinating people and to easily get in touch with renowned researchers from the real-time community. Personally, I will always remember the magic moment of winning the 2nd price for our live demo at the Open Demo Session of Real-Time Techniques and Technologies of the IEEE Real-Time Systems Symposium in Vienna in 2011.

This thesis can also be seen as a result of many years of interactions and discussions with many people. Therefore, before coming to the main body of work, I would like to thank those who accompanied, encouraged, and supported me during this exciting time.

First of all, I would like to express my sincere gratitude to my supervisor Prof. Gerhard Fohler for giving me the opportunity to pursue the Ph.D. at the Chair of Real-Time Systems here in Kaiserslautern. I would like to thank him for his guidance and truly inspiring support on an academic as well as on a human side during the entire period of my studies.

Also, I would like to take the opportunity and extend my appreciation to my committee members: Prof. Steven Liu and Prof. Johan Eker.

Furthermore, I am grateful to Prof. Luca Benini from the University of Bologna for granting access to the MPARM simulator. Special appreciation is also extended to Paolo Burgio from the University of Bologna for helping me setting up MPARM in Kaiserslautern. Likewise, I have to thank Francesco Esposito from the Scuola Superiore Sant'Anna of Pisa for answering many detail questions related to MPARM.

I would like to thank all my friends and colleagues from the Chair of Real-Time Systems: In particular, I thank Ankit Agrawal, Rodrigo Coelho, Gautam Gala, Raphael Guerra, Anand Kotra, Mitra Nasri, Simara Perez, Ramon Serna, Ali Syed, and Jens Theis for the interesting discussions, fruitful collaborations, and all the fun we had.
A special note of appreciation goes to my friend Oliver Marx who always supported me, thank you so much!

In addition, I would like to express my great appreciation to our technician "Scotty" Markus Müller who professionally helped me resolving all kinds of IT-related disasters. He always found solutions to fix problems and to boldly go where no technician has ever gone before.

I am also very grateful to our secretary Stephanie Jung. Always with a smile on her face, she helped to fight German bureaucracy. Without her, it would have been much harder to master all the unbelievably time-consuming and distracting small issues that (un)expectedly appeared out of nowhere. Also, I have to extend my special appreciation to Carmen Vincente-Fess for all the quick and professional help that she provided.

Moreover, I would like to thank the students John Gamboa, Tushar Karayil, Don Kuzhiyelil, and Ivan Shcherbakov for helping me implementing and testing the tools used throughout the different experiments.

Finally, I must admit that I lack the words to adequately express my appreciation and gratefulness to my family. Especially, I would like to thank my uncle for his profound help. Last, but not least, I am deeply indebted to my parents for the invaluable encouragement and the strong and warm support during these years. Thank you all very much!

Stefan Schorr
Kaiserslautern, 14.01.2015

The work presented in this thesis has been partially supported by the European Commission under the European IST-FP7 project *Adaptivity and Control of Resources in Embedded Systems* (ACTORS, FP7/2007/ICT/216586).

Publications

I have authored or co-authored the following publications:

Journal papers

- Enrico Bini, Giorgio Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Karl-Erik Årzén, Vanessa Romero Segovia, and Claudio Scordino, *Resource Management on Multicore Systems: The ACTORS Approach*, IEEE Micro, May 2011.

Conference and refereed workshop papers

- Stefan Schorr and Gerhard Fohler, *Integrated Time- and Event-triggered Scheduling – An Overhead Analysis on the ARM Architecture*, RTCSA2013: Proceedings of the 19^{th} IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Taipei, Taiwan, August 2013.

- Stefan Schorr, Anand Meher Kotra, Gerhard Fohler, Johan Eker, Karl-Erik Årzén, and Vanessa Romero Segovia, *Adaptive Resource Management in the ACTORS Framework – A Live DVB-T/Webcam Demo*, RTSS@Work Demo Session of the 32^{nd} IEEE Real-Time Systems Symposium, November 2011.

- Karl-Erik Årzén, Vanessa Romero Segovia, Stefan Schorr, and Gerhard Fohler, *Reservation-Based CPU Management for Multicore Platforms*, RTiS2011: Real-Time in Sweden, Mälardalen University, Västerås, Sweden, June 2011.

- Karl-Erik Årzén, Vanessa Romero Segovia, Stefan Schorr, and Gerhard Fohler, *Adaptive Resource Management Made Real*, Proceedings of the 3^{rd} Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2011), Chicago, USA, April 2011.

- Karl-Erik Årzén, Vanessa Romero Segovia, Mikael Kralmark, Stefan Schorr, Anand Meher Kotra, and Gerhard Fohler, *Demo: Adaptive Resource Management Made Real*, APRES2011: Proceedings of the 3^{rd} Workshop on Adaptive and Reconfigurable Embedded Systems, Chicago, April 2011.

- Stefan Schorr and Gerhard Fohler, *Online Admission of Non-Preemptive Aperiodic Tasks in Offline Schedules*, ECRTS2010: Proceedings of the Work-in-Progress Session of the 22^{nd} Euromicro Conference on Real-Time Systems, Brussels, Belgium, July 2010.

- Vanessa Romero Segovia, Karl-Erik Årzén, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Johan Eker, and Harald Gustafsson *Adaptive Resource Management for Mobile Terminals - The ACTORS Approach*, WARM10: Proceedings of the 1^{st} Workshop on Adaptive Resource Management, Stockholm, Sweden, April 2010.

Technical Reports

- *Implementing the Slot Shifting Algorithm on the MPARM Simulator*, Stefan Schorr and Gerhard Fohler, Chair of Real-Time Systems, Technische Universität Kaiserslautern, March 2012

- *Simulation Results for Slot-Shifting - Response Times and Acceptance Ratios*, Stefan Schorr and Gerhard Fohler, Chair of Real-Time Systems, Technische Universität Kaiserslautern, March 2014

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time systems are systems that must fulfill twofold constraints, in order to be considered working functionally correct: First, they must process information and consecutively produce a correct output behavior. Second, their interaction with the environment must happen within stringent timing constraints dictated by this environment. In contrast to many other systems, real-time systems are thus not primarily optimized for speed (in the sense of maximized for throughput), but to provide determinism and worst case guarantees.

To ensure meeting their strict timing constraints, real-time systems utilize *real-time scheduling algorithms*. A scheduling algorithm determines the execution order of the jobs of the workload on the processors of the system. There exist many different classification schemes for scheduling algorithms, e.g., depending on the method to prioritize different jobs, or depending on the moment in time when scheduling decisions are made. For this thesis, an important classification of scheduling algorithms is into *time-triggered* and *event-triggered* algorithms. Event-triggered algorithms are based on a set of rules that are used at runtime of the system to make the scheduling decisions. In contrast to this, time-triggered algorithms determine the execution order of jobs offline, i.e., prior to the runtime of the system. We will later in this thesis discuss the advantages and disadvantages of both categories.

There also exist hybrid scheduling algorithms: *Slot shifting* is such an algorithm which combines the benefits of both time- and event-triggered scheduling algorithms. The original algorithm is targeted at distributed systems and works as follows: In a first step, it resolves the complex constraints of a given task set by constructing an offline scheduling table. Using this scheduling table, it provides predictability and deterministic guarantees for the execution of pre-planned tasks at runtime of the system. Additionally, slot shifting provides flexibility to enable the system to react to unforeseen events at runtime, such as job arrivals of aperiodic tasks. Therefore, the algorithm performs an online acceptance test for the individual jobs of firm aperiodic tasks. In case of success, it invokes a guarantee algorithm to feasibly integrate the aperiodic jobs along with the other already guaranteed jobs.

In this thesis, we extend the slot shifting algorithm to feasibly integrate jobs of non-preemptive firm aperiodic tasks. As another contribution of this thesis, we derive—after a careful inspection of the issues that arise with multicore systems—novel global

slot shifting algorithms targeted for multicore systems. The thesis also analyzes and evaluates the low-level properties and the runtime behavior of the algorithms in terms of effectiveness and efficiency in great detail.

Another contribution of this thesis is to show that these slot shifting algorithms offer a suitable solution to the resource management problem on multicore systems. To highlight slot shifting's capabilities, we implemented a slot shifting based logic into an existing generic resource management framework for multicore systems. Our measurements performed on this implementation confirmed the validity and the technical feasibility of the approach.

The remainder of this chapter is organized as follows: First, this chapter starts with a description of the problems that this thesis deals with in section 1.1. After that, section 1.2 illustrates our approach to tackle these problems and lists the contributions of this thesis to the state-of-the-art of real-time scheduling theory. Next, section 1.3 introduces basic terms and principles and presents the related work in this area. Finally, section 1.4 details the outline of the rest of the thesis.

1.1 Problem Statements

Today's real-time systems are found throughout all industries, e.g., in industrial process control, in communication, in entertainment and multimedia, and in safety critical systems. The continuing trend to shrink structures on the silicon die facilitated the advent of modern multicore processors. These multicore processors become more and more prevalent in embedded systems and currently, complete multiprocessor systems-on-a-chip (MP-SoC) are being built. With the implementation and use of real-time systems based upon such systems, new crucial challenges arise on many different levels:

First, from a hardware point of view, multicore systems are highly complex homogeneous or heterogeneous systems. They feature cores of potentially different capabilities designed on top of different instruction set architectures. As the number of cores keeps growing, traditional bus systems that connect the cores are being replaced by costum-tailored complex network-on-chips (NoCs). Furthermore, there is a trend to partially or completely replicate hardware units numerous times to limit contention for scarce hardware units. As a result, hardware analysis of multicore systems becomes extremely challenging. Since concurrently running programs on different cores interfere with each other, providing reliable worst case runtime estimates for compiled source code becomes infeasible in practice. Although the in-depth analysis of multicore systems is out of the scope of this thesis, we believe that the approaches proposed in this thesis are suitable to establish a higher degree of predictability and to provide guarantees to soft real-time applications. Given future processors designed with the focus on deterministic, time-bounded operation, we believe that our approaches are also suitable for the design of hard real-time systems.

Second, from the real-time scheduling point of view, scheduling algorithms are required that improve to use the computational resources of the system. Thus, these scheduling algorithms must feature low overheads to leverage as much computational resources as possible for the applications at runtime. On the one hand, they must pro-

vide a satisfactory level of predictability and worst case guarantees for the tasks. On the other hand, they are expected to provide flexibility to react to aperiodic tasks. Last but not least, it is desirable that these algorithms are able to handle the more and more complex constraints between interacting applications, e.g., end-to-end deadlines of tasks.

Third, from a global, system-wide point of view, just to statically oversee the numerous concurrently running applications on a multicore system, each with different and potentially varying resource requirements, poses a challenge. Moreover, the set of running applications can change dynamically. At any time, some applications might finish their execution and terminate, while other applications may enter the system and request resources. These concurrently running applications are in a constant fight for the scarce resources of the system. Without imposing any restrictions it is impossible to analyze and provide worst case guarantees to hard or soft real-time applications, especially when they themselves feature varying resource needs. For these reasons, there is an urgent need for *resource management* of the limited resources of the platform and to provide reliable worst case guarantees to applications.

In general, resource management aims at fulfilling global optimization objectives. These objectives include, e.g., ensuring minimum resource availability to applications, or tuning of global system parameters, such as maximizing global system performance, or maximizing the globally provided *quality of service* (QoS). As a result of all the aforementioned issues, the design and implementation of an effective and efficient resource management on multicore systems is not a trivial undertaking.

1.2 Contribution of the Thesis

This thesis analyzes the original slot shifting algorithm and proposes an improved guarantee algorithm. Furthermore, it extends the slot shifting algorithm to handle jobs of non-preemptive firm aperiodic tasks. The main contribution of this thesis involves the design, implementation, and evaluation of global multicore slot shifting algorithms. Furthermore, we prove applicability of the slot shifting algorithm to multicore resource management for soft real-time applications. In the following, we detail the contributions of this thesis.

1.2.1 Non-Preemptive Slot Shifting

In chapter 3, we tackle the problem to extend the slot shifting algorithm to feasibly integrate jobs of non-preemptive firm aperiodic tasks at runtime. The main challenge is to derive a method to determine whether or not there exist sufficient resources to feasibly integrate a particular non-preemptive job. Step-by-step, we develop a new acceptance test; the basic idea of which is similar to that of the original acceptance test: It checks interval after interval, starting from the current interval of time up to the interval hosting the aperiodic job's deadline for consecutively available free slots in the schedule. While checking, our algorithm identifies these jobs whose shifting results in more consecutive slots for non-preemptive execution.

The main advantage of our algorithm is that it does not require to recalculate a completely new offline scheduling table. Instead, it only modifies the existing table. Furthermore, our algorithm has low memory requirements and a runtime complexity of $O(n^2)$. Additionally, we discuss the possibility to tune the guarantee algorithm to either improve the response time of non-preemptive jobs or to delay their execution in order to increase flexibility for potentially arriving future aperiodic jobs.

1.2.2 Multicore Slot Shifting

Chapter 4 focuses on the design of global slot shifting algorithms for multicore systems. A known fundamental result of real-time scheduling theory is the non-existence of optimal online multiprocessor scheduling algorithms for aperiodic tasks. Hence, we focus on designing global slot shifting algorithms with the twofold goal of achieving satisfactory results while featuring acceptable runtime overheads.

We discuss the challenges of different methods to tackle the problem. The first challenge arises when local acceptance of an aperiodic job fails. In that case, the corresponding core needs to quickly find another suitable core for potential acceptance of the aperiodic job there. Another challenge involves the decision on how to store the offline scheduling table: in a distributed or in a global fashion. Further challenges include the minimization of data exchange and thus synchronization between the cores to avoid one core overly hindering the progress of others.

Finally, we propose two global algorithms: a spare-capacity-based and a negotiation-based version of the slot shifting algorithm. Both algorithms employ different heuristic-based approaches to determine a suitable core for an aperiodic job that cannot be guaranteed locally. To avoid contention, both global algorithms store the corresponding offline scheduling table data locally per core; and both algorithms are designed to minimize data exchange in order to reduce overheads.

1.2.3 Evaluation of the Slot Shifting Algorithms on Multicore Systems

The succeeding chapters 5 and 6 perform a multitude of experiments to evaluate the original and the previously presented global slot shifting algorithms. The experiments aim at providing a holistic view of the properties and the runtime behavior of the algorithms.

Chapter 5 employs a cycle-accurate MP-SoC simulator to evaluate slot shifting in terms of efficiency. The experiments analyze the runtime requirements of different sub functions of the algorithms. As a result of these experiments, we propose an improved, i.e., faster, guarantee algorithm that shortens the deadline of the aperiodic jobs, thus named *SDL*. Consequently, SDL also yields improved response times of the aperiodic jobs.

Chapter 6 uses a different, Linux-based implementation to evaluate the slot shifting algorithms in terms of effectiveness. For each of the different experiments, we create a vast amount of random task sets. Then, we schedule the same task sets with different slot shifting algorithms and—as a comparison metric—also with EDF with background

processing of the aperiodic jobs. We determine the maximum achievable acceptance ratio within 95% confidence intervals, analyze the average number of performed acceptance tests, and measure the resulting response times of aperiodic jobs. The experiments underline that using SDL significantly improves the response times of the aperiodic jobs, for all slot shifting algorithms. This improvement comes at zero cost, since the acceptance ratio remains virtually constant. Furthermore, the experiments show that in terms of achieved acceptance ratio for the aperiodic jobs, the global slot shifting algorithms always outperform the original, partitioned slot shifting algorithm, which in turn clearly surpasses EDF with background processing of the aperiodic jobs.

1.2.4 Resource Management

Our final contribution lies in the field of adaptive resource management for soft real-time applications on multicore systems. Today's multicore systems concurrently execute many applications with different resource requirements. Without active management of the shared resources of the system, applications would fight for the scarce resources, leading to contention and starvation of applications, and thus real-time guarantees cannot be given. Adaptive resource management is targeted for systems in which applications offer multiple modes of execution with different resource requirements and offering different QoS to the user. Adaptive resource management is able to react to changes such as terminating or newly entering applications. Additionally, it can reclaim unused resources to avoid costly over-provisioning of the system.

The contribution of this thesis is to prove that slot shifting can be employed in this context to provide deterministic guarantees to soft real-time applications while allowing for runtime flexibility. To demonstrate the validity and the technical feasibility of the approach, we implemented a slot shifting-based logic into the resource manager of an adaptive generic resource management framework for multicore systems.

1.3 Background and Related Work

The analysis and implementation of today's real-time systems embodies a large amount of complexity. In order to analyze these systems, researchers abstract from the specific properties of hardware and software to obtain less complex models of the real-world systems. Once these models have been derived, methods from real-time scheduling theory are applied to them to provide timing guarantees for the system.

The purpose of this section is to introduce some of the basic terms of the real-time scheduling theory and to establish the needed background underlying this thesis. The initial sub sections introduce the most important terms and abstractions that form the task and the system model. Then, there follows an overview of the real-time scheduling theory that is fundamental to this work, with a special focus on event- and time-triggered scheduling and on multiprocessor scheduling. After this, there follows a sub section on resource management in real-time systems, in which we define the terms required for chapter 7.

1.3.1 Basic Terms

One of the most basic terms used in real-time scheduling theory is that of a *task*. A task is an abstraction of a piece of software that implements a basic functionality in a real-time system. Over the years, different *task models* that impose different assumptions about the nature and behavior of the tasks have been proposed in literature.

The most fundamental and influential task model in real-time scheduling theory is the so called *periodic task model* introduced by Liu and Layland [4]. In this task model, every task represents work that needs to be performed periodically. An example for such a task is the software in an airplane's computer that is periodically triggered during the operating time to read sensor values, e.g., speed and altitude, and to adjust actuators correspondingly to ensure correct and safe flight conditions. Individual invocations of a task are called *jobs* or *instances* of the task.

Liu and Layland based their model on the following, basic assumptions: A task consists of a possibly never-ending sequence of instances/jobs. All tasks are strictly periodic, i.e., their jobs are periodically released at fixed, equidistant moments in time. All jobs of a task feature the same known upper bound on their execution time, the *worst case execution time* (WCET). Furthermore, they feature deadlines which are assumed to be equal to the period of the task. Another assumption is that tasks are completely independent from each other. Moreover, jobs do not suspend themselves for any reason and the overhead to manage and execute, i.e., to *schedule* the individual jobs of the different tasks is considered negligible. Sha et al. pointed out that Liu and Layland also make the implicit assumption that the system features only a single processor [5].

In other words, tasks in the periodic task model are described by a pair (C, T), where C is an upper bound on the execution requirement of the task (the worst case execution time). T is the period of the task, i.e., the time span between two consecutive invocations of the task. The individual jobs do not feature an explicitly given deadline. Instead in Liu and Layland's model, the deadline is implicitly assumed to be equal to the period of the task, i.e., jobs need to finish their execution before the arrival of the next job of the same task.

In real-time scheduling theory, deadline constraints of task sets thus are classified as *implicit*, when the deadlines of the tasks are equal to their periods. A task set features *constrained* deadlines, when the deadlines of the tasks are less than or equal to their periods. And if none of the former conditions applies, a task set is said to have *arbitrary* deadline constraints.

When the jobs of all tasks are simultaneously released, i.e., with the same offset, the term *synchronous* task set has been established. If not mentioned otherwise, task sets are usually assumed to have a offset of zero time units.

All jobs are assumed to be fully *preemptive*, i.e., higher priority jobs can interfere at any time and delay the execution of lower priority jobs. Tasks whose jobs cannot be preempted are called *non-preemptive* tasks.

The *utilization* of a task τ_i is defined as $U_i = \frac{C_i}{T_i}$ and the utilization of the entire task set is defined as $U = \sum_{i=1..n} U_i$.

Another task model is the *sporadic task model* [6], which is a relaxation of the periodic task model. Instead of a strict period, this model advocates a minimum interarrival

time between consecutive occurrences of jobs, i.e., job arrivals are allowed to happen less frequently. Many more models have been proposed to cover different scenarios, e.g., the generalized multiframe (GMF) model [7], the intra-sporadic task model [8], the gravitational task model [9], DAG based models [10, 11, 12], or models based on timed automata as in [13]. The interested reader might refer to the surveys in [14] and [15].

Apart from the tasks modeled as described above, there exists another type of tasks: Tasks that feature completely unknown arrival patterns. These tasks are called *aperiodic tasks* and are—like periodic tasks—assumed to consist of a possibly infinite sequence of jobs without any restrictions on their interarrival time [16]. If aperiodic tasks feature deadlines, they are referred to as *firm* aperiodic tasks, otherwise as *soft* aperiodic tasks.

Since their arrival cannot be predicted, the instances of firm aperiodic tasks require individual treatment at runtime of the system in order to provide timing guarantees. Systems handling firm aperiodic tasks typically follow a twofold aim: First, to guarantee their timely execution and second, to minimize their *response time*, i.e., the time span between the arrival of an aperiodic job and the end of its execution.

Our approaches described in this thesis are based on the Liu and Layland task model: We handle preemptive tasks with constrained deadlines. These tasks may be subject to complex constraints, e.g., precedence relations or end-to-end deadlines, expressed by precedence graphs. Furthermore, our approaches handle soft and firm non-preemptive and preemptive aperiodic tasks.

1.3.2 System Model

The *system model* specifies the type of a real-time system, focusing on its main properties, such as number and speed of processors, and abstracting away unnecessary details, such as register configuration or power consumption of the processor. Further, the system model specifies whether it is a *hard* or a *soft* real-time system. Hard real-time systems must obey all timing constraints of all tasks of the given task set in order to work correctly and even a single deadline miss cannot be tolerated. Soft real-time systems tolerate occasional deadline misses up to a limited extent. Examples for the former category are, e.g., the airbag controller in a car, or the automatic aircraft flight control system in an airplane. An example for the latter category is the graphical user interface of a standard word processing program, in which occasionally missing to update the screen for the split of a second will be tolerated.

A common misconception is that people believe that this classification of a real-time system implies the impact of the consequences of deadline misses. This is however not the case; the consequences of a malfunction and the classification as hard or soft real-time system are completely orthogonal dimensions. On the one hand, while a deadline miss of most hard real-time systems might endanger human beings or worse, lead to loss of human lives, there exist many hard real-time systems whose failure will not lead to severe consequences. For example, every DVD recorder is a hard real-time system, whose timely misbehavior will only annoy the owner but typically not endanger his life. On the other hand, it is possible that the malfunction of a soft real-time system has severe consequences. An example for such a system might be an industrial control sys-

tem that monitors the temperature of chemicals stored in a tank. When such a system misses deadlines once in a while, it still works satisfactory. The physical world has a certain inertia that slows down state changes of the system, in this case rapid changes of the temperature of the chemicals in the tank. The more deadlines are missed, though, the more the state of the system assumed by the control system might differ from the actual conditions in the tank—until the system eventually fails when too many deadlines were missed. Although such a system is classified as a soft real-time system, the consequences of its failure to work correctly might be catastrophic. Obviously, this kind of soft real-time systems is usually designed such that it switches to a safe mode in case of too many deadline misses.

1.3.3 Processor Model

Early computer systems consisted of a set of chips surrounded by peripheral devices and memories that altogether established the capability to perform computations and to execute programs. The heart of these chips was the central processing unit (CPU) or processor. This processor can be imagined as the brain of the system: "The basic cycle of every [processor] is to fetch an instruction from memory, decode it to determine its type and operands, execute it, and then [proceed with] subsequent instructions" [17].

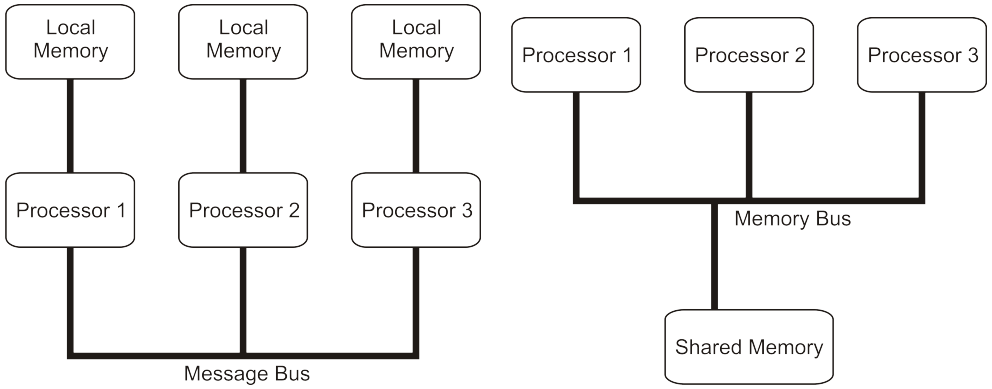
Today's systems incorporate the functions of multiple processors on a single chip: a *multiprocessor* (chip). "A multiprocessor consists of multiple, independently controlled processing units that communicate via a processor interconnect" [18]. Nowadays, there exists a wide variety of multiprocessor systems that can be classified, e.g., as either *homogeneous* or *heterogeneous* depending on the processor type(s) within. Multiprocessor systems of the former category consist of a set of identical replicas of processors that may or may not run at the same speed. The latter category is composed of different, often specialized processors that may feature completely different architecture types. Thus, they often use different instruction sets and might operate at the same or independent clock frequencies.

Another way of categorizing a multiprocessor system is by the nature of their interconnect:

- A *distributed-memory multiprocessor* system consists of locally separated, independent processors and each processor has access to its local memory. Although all processors are connected via a message bus system, there is no direct way for a processor to access non-local memories, see Figure 1.1a.

- A *shared-memory multiprocessor* consists as the name suggests of independent processors sharing one common memory via a memory bus, see Figure 1.1b. Shared memory multiprocessor systems can be further divided into *uniform memory access* architectures (UMA) and *non-uniform memory access* architectures (NUMA): In the former category, the access time from any processor to any random memory location is the same. The latter category features a more complex memory hierarchy of local and non-local memories which causes access times to differ depending on origin and destination of requests.

Independent of the above classification, the individual processors may operate at identical or unrelated clock frequencies.



(a) *A distributed-memory multiprocessor.* **(b)** *A shared-memory multiprocessor.*

Figure 1.1: *Illustration of distributed- and shared-memory multiprocessors, inspired by [19].*

In the last couple of years, other terms have been established: "In a *multicore* design, multiple (mostly) independent processing cores are manufactured on a single integrated circuit chip to exploit increases in transistor density. From a scheduling point of view, "multicore" is thus simply a particular way of implementing multiprocessors" [19]. Following this notion, we will from now on use the terms multicore and multiprocessor interchangeably.
As transistor densities are still increasing, more and more of the functionality of the system is integrated into a single chip. Thus today, there exist complete *multiprocessor-systems-on-a-chip* (MPSoC) that—additionally to the processors—feature a large variety of components, e.g., memory hierarchies, sensors, and interfaces to different bus systems.

The work presented in this thesis assumes homogeneous, shared memory multiprocessor systems with uniform memory access times, operating at identical clock frequencies.

1.3.4 Real-Time Scheduling

Scheduling refers to the process of making and enacting the decision which jobs from a set of ready jobs to execute next. A set of rules that defines the steps needed to determine the next jobs for execution is called *scheduling algorithm*. Scheduling algorithms can be classified in many ways, e.g., whether they assign fixed or dynamic priorities to jobs of a task, or whether they are uni- or multiprocessor algorithms, or whether they allow for preemptions or not. The entity that enforces these scheduling algorithms is called a *scheduler*. During its operation, the scheduler produces a *schedule*, i.e., a job execution sequence. In other words, a schedule specifies the exact moments in time of the assignments of the individual jobs to the processor(s).

If all the specified timing constraints of the task set are met, a schedule is termed *valid*, or *correct*. Further, a task set is called *feasible* with respect to a particular system, if

there exists at least one correct schedule. A task set is called *schedulable* with respect to a given system and according to a given scheduling algorithm, if a correct schedule can be constructed using this specific algorithm, i.e., such that in the resulting schedule no job violates its timing constraints. Moreover, a scheduling algorithm is called *optimal*, if the scheduling algorithm produces a correct schedule for all task sets that are feasible on a particular system. Note that typically, a given optimal scheduling algorithm is only optimal for a certain class of scheduling algorithms. For example, while earliest deadline first (EDF) is an optimal scheduling algorithm for the preemptive uniprocessor class of scheduling algorithms [4], it is not optimal for non-preemptive uniprocessor scheduling when idling is allowed, or for preemptive multiprocessor scheduling [19].

Much effort has been spent to derive *schedulability tests*. The purpose of these tests is to determine for a given task set and a particular platform the schedulability with a specific scheduling algorithm without the need to construct a valid schedule. We distinguish the following types of schedulability tests:
A *necessary* schedulability test imposes a condition on task sets that must be fulfilled, i.e., failing this test identifies definitely not schedulable task sets. However, passing this test does not reliably confirm schedulability.
Passing a *sufficient* schedulability test correctly identifies a task set as schedulable. Nevertheless, sufficient schedulability tests are pessimistic and thus may reject some in fact schedulable task sets.
If a schedulability test returns a positive result on all schedulable task sets and rejects all unschedulable task sets, it is called an *exact* schedulability test.

For uniprocessor scheduling with periodic tasks, schedulability tests are based on the *critical instant* of a task [4]: The moment in time when a job of this task will have the largest response time, i.e., when it is released at the same time as the jobs of all higher priority tasks. The response time of a task, or more precisely of a job, is the time span between its release time and its finishing time.

It is known that with increasing complexity of the underlying task model, feasibility analysis becomes a computationally expensive, or even intractable operation [15]. Even for uniprocessor algorithms, there exist pathological cases, see the section on non-preemptive scheduling below.

1.3.5 Classification of Scheduling Algorithms

There are many different ways to classify different scheduling algorithms; we list the most important categories in the following.

1.3.5.1 Preemptive vs. Non-Preemptive Scheduling

Schedulers can be classified according to their decision mode: If a scheduler selects a job to run and, under any circumstances, waits for this job to complete before any other action from the scheduler takes place, this scheduler is called a *non-preemptive* scheduler. If a scheduler is free to interfere with a currently running job at any time, this scheduler is classified as *fully preemptive* scheduler.

It has been proven that deciding whether or not a given concrete periodic task set, i.e., a task set together with arbitrary but known release offsets, is non-preemptively schedulable upon a single processor is intractable—NP-hard in the strong sense [20]. In other words, unless P = NP, this problem cannot be decided by any polynomial runtime algorithm [21]. Additionally, the authors of [20] also showed that non-preemptive EDF is optimal in the class of non-preemptive work-conserving, i.e., non-idling, uniprocessor scheduling algorithms for both periodic and sporadic tasks. More recent research has shown "previous scheduling algorithms and schedulability tests to be too conservative especially when applied on harmonic task sets"; i.e., task sets that feature tasks with periods that are integer multiples of each other [22].

1.3.5.2 Fixed Priority vs. Dynamic Priority Scheduling

Another classification of scheduling algorithms is established on the paradigm by which priorities are assigned to the tasks. If each task is assigned a fixed priority that does not change at runtime from job to job of the same task, then this is called *fixed (task) priority scheduling* (FPS). The rate monotonic (RM) and the deadline monotonic (DM) scheduling algorithms are examples for such an algorithms: The RM algorithm assigns task priorities according to the periods of the tasks, i.e., the smaller the period, the higher the priority. Analogously, the DM algorithm assigns task priorities according to the relative deadlines of the tasks, i.e., the smaller the deadline, the higher the priority. RM is optimal among the preemptive FPS algorithms for synchronously released task sets with implicit deadlines. In other words, if a given synchronously released task set with implicit deadlines is schedulable according to any FPS algorithm, then RM is also able to create a valid schedule. For synchronously released task sets with constrained deadlines, DM scheduling is optimal [5]. However, "under the assumption of specified offsets the rate and deadline monotonic algorithms are no longer optimal" [5], i.e., RM and DM are not optimal for asynchronous tasks sets[1].
For RM scheduling, a necessary schedulability test for a given task set with implicit deadlines that can be performed with linear runtime complexity has been proposed by Liu and Layland [4]:

$$\sum_{i=1}^{n} U_i \leq n * (\sqrt[n]{2} - 1) \; \forall \tau_i \tag{1.1}$$

with n the number of tasks. For large n, the maximum worst case utilization bound becomes $\ln(2) \approx 0.69$. An even tighter necessary schedulability test with linear runtime complexity, the *hyperbolic bound* [24], is given by:

$$\prod_{i=1}^{n} (U_i + 1) \leq 2 \; \forall \tau_i \tag{1.2}$$

The exact schedulability test for a given task set under RM can be performed in pseudo-polynomial runtime complexity, using *response time analysis* [25].

[1] In that case, Audsley's priority assignment algorithm is known to be optimal [23].

If the priority of jobs of the same task may change over time or from job to job, this is called *dynamic task priority scheduling*. Such scheduling algorithms can again be subdivided into two classes: In *fixed job priority scheduling*, the priority of the individual jobs is fixed but can differ for distinct jobs of the same task. An example for such a scheduling algorithm is, e.g., EDF. Under EDF scheduling, jobs are prioritized based on their deadline: The job with closest deadline is assigned highest priority and thus scheduled first. As already mentioned, EDF is optimal among all preemptive uniprocessor scheduling algorithms for task sets with implicit or constrained deadlines. An exact schedulability test for a given task set with implicit deadlines can be performed with linear runtime complexity:

$$\sum_{i=1}^{n} U_i \leq 1 \; \forall \tau_i \tag{1.3}$$

The exact schedulability test for a given task set with constrained deadlines can be performed with pseudo-polynomial runtime complexity using the *processor demand approach* [26, 27].

In *dynamic job priority scheduling*, the priority of the individual jobs may change over time as, e.g., in least laxity first (LLF) scheduling [28] or in proportionate fair (PFAIR) scheduling [29].

1.3.5.3 Event-Triggered vs. Time-Triggered Scheduling

Another way of classifying schedulers is based on the mechanism that enables the scheduling process. In *time-triggered* scheduling, all activities of the scheduler are triggered by the progression of time, i.e., scheduling decisions occur at pre-defined moments in time. The scheduler uses a scheduling table that has been computed prior to the runtime of the system to base all scheduling decisions on. Time-triggered scheduling is also called *offline* scheduling.
In *event-triggered* scheduling, events, e.g., task releases, task completions, or external interrupts initiate actions of the scheduler. Event-triggered scheduling is also called *online* scheduling, since all scheduling decisions are made at runtime of the system, based on a set of rules which altogether form the *scheduling algorithm*.

One advantage of time-triggered scheduling is the inherent temporal isolation among the individual jobs of the tasks. Potential job overruns, do not harm schedulability of other jobs, as the scheduler regularly interferes. Such overruns occur when a job executes for longer than its predicted WCET. Another major advantage of time-triggered scheduling is its predictability: "Time-triggered architectures are inherently more predictable than event-triggered architectures" [30], since their reactions to the environment are more regular. Moreover, since in time-triggered architectures every possible action is pre-planned offline, certification is much simpler than in event-triggered architectures. One drawback of time-triggered systems is that if system designers fail to offline specify actions, then the system is not capable to perform the intended actions at runtime. This pre-planning requires prior knowledge of all potentially occurring future runtime events. Another major drawback inherent to time-triggered systems is that adding one single task might require a re-design of the complete schedule of the system.

Event-triggered systems typically react to unforeseen events, e.g., the arrival of aperiodic tasks, more flexible than time-triggered systems. This increased flexibility is not free of charge and typically comes at cost of decreased predictability: While offline guarantees can be provided that, e.g., jobs execute prior to their deadline, it is hard to predict the exact moments in time when they actually execute. Furthermore, this increased flexibility adds runtime overheads: The resource utilization of event-triggered systems is much better on low or average load conditions than that of a comparable time-triggered system. "In peak load scenarios the situation can reverse, since the time available for the execution of the application tasks is reduced by the increasing processing time required for executing the interrupt handling, buffer management, synchronization, and scheduling algorithms" [30].

Interferences by the interrupt handlers or by the operating system handling its internal state (buffers, etc.) can be substantial in event-triggered systems. Depending on the particular system, these interferences can trigger more worst case execution time variations for a deadline constrained real-time task than that caused by the input data dependency of the task itself [30, 31].

Another major issue of event-triggered systems is that potentially occurring overruns of jobs might affect schedulability of the complete system. Furthermore, during the design of event-triggered systems there exists the non-trivial problem to determine suitable buffer sizes, since the estimation of the worst case demand is subject to probabilistic methods [30].

1.3.5.4 Multiprocessor Scheduling Algorithms

After having discussed the established uniprocessor scheduling algorithms as, e.g., RM or EDF in section 1.3.5.2, this section focusses on multiprocessor scheduling algorithms.

"Multiprocessor real-time scheduling is intrinsically a much more difficult problem than uniprocessor scheduling" [32]. Already in the year 1969, Liu noted:

"Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors" [33].

In multiprocessor systems, the scheduling approaches can be categorized into two major categories based on the degree at which tasks are permitted to migrate between processors: *partitioned* scheduling and *global* scheduling. Additionally, there exist *hybrid* scheduling approaches of the aforementioned categories.

In partitioned scheduling, all tasks of the task set are pre-assigned to specific processors before the system is started. At runtime of the system, tasks release their jobs only on the processor they have been assigned to and jobs are not allowed to migrate to other processors of the system. This scheduling approach offers some obvious advantages: Scheduling of the jobs on the individual processors becomes a simpler and optimally solvable uniprocessor scheduling problem. Additionally, this simplifies the

management of the jobs of the job set. Partitioned scheduling does not suffer from migration overheads since jobs are not allowed to migrate among the processor. Furthermore, if a job runs longer than expected, this overrun only affects a single processor, thus does not harm any job on other processors. These advantages come at the price of the following disadvantages: The partitioning and allocation of the tasks to the processors is similar to a bin-packing problem. This category of problems is known to be NP-complete in the strong sense [34]. Hence, fast but suboptimal heuristic approaches, e.g., Best-Fit or First-Fit heuristics, are applied[2]. Approaches to solve this partitioning problem further include simulated annealing [35], branch and bound [36], and integer linear programming [37]. Another disadvantage is that partitioned scheduling is not *work-conserving*: It is possible that one or multiple processors run idle, while there exist ready jobs waiting for execution on other processors. Another drawback is that the worst case utilization bound for periodic task sets with implicit deadlines scheduled by any partitioning algorithm on M processors is not better than $\frac{M+1}{2}$ [38]. The most important disadvantage is the non-optimality of partitioned scheduling: There exist task sets that cannot be scheduled successfully using any partitioned scheduling algorithm.

In global scheduling algorithms, all ready jobs are enqueued in a single ready queue that is managed by the global scheduler. Depending on when these decisions are enacted, global approaches are further divided into two classes: In *restricted migrative scheduling*, the tasks can only migrate at job boundaries, while in *fully migrative scheduling*, the tasks can migrate at arbitrary moments in time, i.e., also in the middle of job execution.

Global scheduling offers the following advantages over partitioned scheduling: There is no initial need to partition the task set, i.e., there exists no bin-packing-like problem that needs to be solved. This allows to easily add (or remove tasks) tasks to the system by adding them to the global ready queue. Global scheduling generally features less context switches and preemptions, because the scheduler will only preempt a job when there are no processors idle [39]. Freely available capacities on one of the processors can be reclaimed by the other processors by migrating one or multiple tasks. If a job of one of the tasks overruns its time budget, the system can react by migrating jobs away. The most important advantage is that there exist optimal global scheduling algorithms, when the scheduler is fully migrative and allows for dynamic job priorities. Examples for such optimal multiprocessor scheduling algorithms for periodic task sets with implicit deadlines are proportionate fair scheduling (PFAIR) [29] and largest local remaining execution time first scheduling (LLREF) [40]. These algorithms feature a worst utilization bound which equals the capacity of the multiprocessor system and thus theoretically allow to make use of the full computational capacity of the system. Nevertheless, PFAIR suffers in practice from very high preemption/migration overheads that occur every time a scheduling decision is made [32].

A common issue to global algorithms is the so called "Dhall-Effect" which limits the worst case utilization bound to 1 in the presence of high utilization tasks [41]. Another common issue is the need for a global lock, since "no other approach has been proposed to date that provably ensures the predictability of migrations and preemptions" [19].

[2] A performance comparison of different suggested heuristic approaches can be found in Table 3 in [32].

This bottle-neck causes contention among the processors and enforces synchronization when scheduling decision are made in time quanta based algorithms such as PFAIR [32].

Additionally, researchers analyzed hybrid approaches: In *clustered* scheduling, processors are grouped to clusters. At runtime, global scheduling algorithms are applied inside each cluster, the applied scheduling algorithm, however, may vary from cluster to cluster. Clustered scheduling can allocate tasks such that they can make use of cache affinity. As a consequence, less pessimistic worst case execution time bounds can be established.

In *semi-partitioned* scheduling, similar to partitioned scheduling, some (or all) tasks are pre-assigned to one or multiple processors. At runtime of the system, migrations can occur; an example for a semi-partitioned scheduling algorithm is, e.g., EKG scheduling [42].

The survey paper of Davis and Burns gives a good overview of the status and the open issues of multiprocessor real-time scheduling theory. "Global, clustered, and semi-partitioned approaches to multiprocessor scheduling offer potential solutions for future complex high-performance real-times systems; however, few results can be identified in these areas that are ready to be transferred into industrial practice" [32].

Another recent scheduling approach is used in *arbitrary processor affinities* (APA) scheduling: APA "permits the specification of migration strategies that are more flexible and less regular than those considered in the literature to date" [43]. It assigns for each task individually a subset of processors to schedule this task on.

In the *limited migrative model* (LMM), tasks are pre-assigned to processors similar to semi-partitioned and APA scheduling. In contrast to traditional scheduling approaches, release and migration decision are made by the tasks themselves by means of agreement protocols. Hence, there is no need for a central scheduler entity [44].

1.3.6 Scheduling of Aperiodic Tasks

In [45], Hong and Leung have shown that there exists no optimal multiprocessor online scheduling algorithm for aperiodic tasks. This result has been extended by Dertouzos and Mok: Even if the worst case execution times (WCETs) are known, no such algorithm exists [46]. Furthermore, in [47] Fisher has shown that there also exists no optimal multiprocessor algorithm to online schedule sporadic task sets with constrained or arbitrary deadlines.

For uniprocessor scheduling, there exist many different approaches to handle aperiodic jobs: Independent of the chosen scheduling model, the most simplistic approach to handle jobs of tasks with unknown arrival patterns is to employ background processing, in which the scheduler selects an aperiodic job to run, whenever no other job is ready to be selected.

Different algorithms have been proposed to improve the responsiveness of the jobs of aperiodic tasks and to provide deterministic guarantees:

Server algorithms for fixed priority scheduling [48, 49, 50], as well as for dynamic priority scheduling [51, 52, 53], aim at reserving a fraction of the processor bandwidth

to the aperiodic jobs. Therefore, server algorithms introduce an additional periodic task, the server task, into the schedule. The main drawback of this approach is that a substantial amount of the CPU utilization might be reserved for future aperiodic jobs that will not necessarily arrive. Another drawback is that using the server algorithm can result in a lower schedulability bound of the system. To the best of our knowledge, it has not been shown how to combine server algorithms with arbitrary time-triggered scheduling tables.
One of the server algorithms, the *constant bandwidth server* (CBS) [54] features some interesting properties. In any given server period, it provides a constant bandwidth, i.e., a constant fraction of the processor time, for the execution of aperiodic jobs, hence its name. In case of early completions of aperiodic jobs, the CBS is able to reclaim unused spare time. Moreover, the CBS does not rely on the worst case execution time of the individual aperiodic jobs. Hence, a CBS can be used even when exact worst case execution times are hard to obtain, overly pessimistic, or unknown. The main advantage of the CBS is that it provides *temporal isolation* among the individual jobs of the system. In other words, an overrun of an aperiodic job will not impact on the schedulability of any other job.

The *slack stealing* algorithm [55] employs another method to handle aperiodic tasks: Offline, it calculates the maximum time by which the jobs of periodic tasks can be delayed before missing their deadlines, the so called slack value. At runtime, when an aperiodic job arrives, slack stealing delays periodic jobs and prioritizes aperiodic jobs to improve their responsiveness. An optimal version, that updates the slack values online has been presented, with the drawback of unacceptable time overheads [56].

The work presented in this paper is based on the *slot shifting* algorithm which combines the benefits of both time- and event-triggered scheduling for distributed systems [1]. Slot shifting resolves the complex constraints of a set of offline tasks by constructing an offline scheduling table. Similar to the aforementioned slack stealing algorithm, slot shifting expresses the leeway of tasks in this derived offline schedule by *spare capacities*. At runtime of the system, slot shifting performs acceptance tests for the individual jobs of aperiodic tasks and integrates them feasibly into the schedule. In [57], slot shifting has been extended to handle sporadic tasks and to reclaim unused resources for sporadic tasks.
Luo and Jha extend the basic slot shifting algorithm to support multi-rate periodic tasks graphs and aperiodic tasks in homogeneous and heterogeneous distributed systems [58]. Their online scheduler supports resource reclaiming, dynamic voltage scaling, and power management.
M. van den Heuvel et al. integrated slot shifting into μC/OS-II, a commercial real-time operating systems (RTOS). The authors added a mechanism for resource reclaiming and rudimentarily determined the runtime overheads. Their work is limited to unicore architectures and periodic offline tasks [59].
Slot shifting can also be used in the field of mixed criticality scheduling. In [60], Theis proposes the use of slot shifting's spare capacities to adapt the selection function and thus guarantee HI-criticality jobs without the need of mode changes.
We extended slot shifting to support the feasible integration of non-preemptive aperi-

odic tasks into the schedule at runtime [61]. Further, in [62] we performed an overhead analysis of slot shifting and showed the runtime costs of integrating aperiodic jobs to be acceptable. In this thesis, we extend this analysis to partitioned as well as global slot shifting on multicore architectures. We analyze these algorithms with various metrics in terms of effectiveness and efficiency. Additionally, we show how slot shifting can be used to perform resource management in a generic, adaptive resource management framework.

1.3.7 Resource Management

Today's embedded systems are based on multicore platforms that concurrently run a variety of applications. Many of the individual applications run independently of the other applications of the system. Hence, a single application might, since it is unaware of the existence and of the requirements of the other applications, reserve or consume too much resources so that the performance of other applications decreases. For the end user, this potentially results in an unacceptable drop of the perceived overall QoS provided by the system. The same reasoning applies to malicious or erroneous applications: Blocking of one or multiple resources potentially leads to starvation of other essential applications. A single application could, e.g., monopolize the processor all the time, or drain the battery of a portable device and hence, render the complete system virtually useless.

For these reasons, the access to the scarce resources of such platforms must be managed. An active *resource management* establishes a notion of fairness among the competing applications and resolves the aforementioned conflicts. Furthermore, it aims at optimizing the overall QoS provided by all applications of the system and copes with misbehaving applications. Adaptive resource management enables systems to work reliably even when the applications and their exact resource needs are unknown at design time, or when they are subject to fluctuations. Many multimedia applications for example feature highly variable resource requirements. Furthermore, resource management is the basis for proficient system-wide energy management.

Nollet et al. were the first to give an overview of the design space for resource managers and to classify the different implementation approaches [63]: First, the authors identify a "hardware versus software design axis". A resource manager by its very nature must interface with the low level hardware services of the platform to enact and enforce its decisions. Hence, the idea to implement parts of its runtime mechanisms in hardware suggests itself. This offers the advantage of reducing the runtime overheads on the processor: If the runtime mechanisms of the resource manager are performed on independent hardware, less system resources are blocked by the resource manager itself and are thus available to the applications.

Another categorization is given by the number and organization of the resource manager entities in a particular system: The resource management can be performed by a central resource manager or by multiple distributed entities. If a system hosts multiple resource managers, they can be classified as *non-cooperative*, if each resource manager independently manages its resources without aiming at a common goal. There also exist

cooperative resource managers, i.e., the individual entities make their own decisions, but collaborate in order to optimize the overall system towards to a common goal.
In a *master-slave configuration*, one processor executes a resource manager that monitors the state of all other processors of the system and assigns the workload to them. On the one hand, this implementation style is simple and efficient. On the other hand, the master presents a potential bottleneck if it fails to assign sufficient work to the slaves. Moreover, this configuration also poses a greater risk to the stability of the system, as it constitutes a single point of failure.
The *separate supervisor configuration* offers a solution to these drawbacks. Every processor executes its own resource manager that has its own data structures and enforces its own decisions. This configuration is scalable and, in case of failure, additional mechanisms can be implemented to ensure that only a single processor is affected, or that other processors take over. The disadvantages that come with this configuration are the increased need of memory due to duplication of data structures on each processor and the unavoidable synchronization penalties to achieve global resource management goals.
In a *symmetric configuration*, every processor individually hosts its own resource manager whose data structures are globally shared. While this approach offers the highest degree of flexibility, it also suffers from some issues: First, the underlying system must facilitate an efficient way to access the shared data structures, e.g., via a shared memory. Second, due to practical implementation issues the symmetric configuration is the most difficult configuration to implement efficiently, since the access to shared data structures can create bottlenecks. Finally, the scalability is limited and lies between the master-slave and the separate supervisor configuration.

A different classification scheme is given by the degree of adaptivity of resource managers: Resource managers can be fine-tuned between a very generic or domain specific implementation. Nollet et al. distinguish between *design-time adaptation*, when resource managers are tuned during the design phase of the system and the applications, or *runtime adaptation* when this adaptation happens at runtime of the system.

Apart from these classification schemes, Nollet et al. also classify the resource assignment algorithms of the resource managers into various categories, see Figure 1.2. There exist static and dynamic algorithms, that again can be sub divided according to various criteria, i.e., whether they achieve optimal or sub-optimal results, whether they are distributed or not, or whether they are cooperative or not, etc.

Real-time resource management covers a large field offering many different challenges. The actual methodology that the resource managers are based on varies depending on the type of resource that is managed.

To manage the processor time assigned to the individual processes, different approaches exist: The most static and most simplistic approach is the use of a time-triggered scheduler that relies on a fixed table which lists the activation order and time span that the individual jobs shall run on the particular processors. Apart from the time-triggered approach, different online algorithms can be used to enforce temporal isolation among the individual jobs on the one hand, and to allow for more flexibility at

runtime than in a purely time-triggered system, on the other hand. In this thesis, we focus on management of the processor time for the individual applications running on multicore systems. Throughout this thesis, we will present and analyze the use of the slot shifting algorithm on multicore architectures. In chapter 7, we propose to use slot shifting in the context of adaptive resource management. We modify a generic resource management framework to incorporate a slot shifting driven management logic. The aim is to show the feasibility of using our slot shifting-based algorithms to tackle the resource management problem on embedded real-time systems.

Even when processes are temporarily isolated from each other, modern processors offer many features that create "holes" in this isolation: Allmost all modern processors feature a cache hierarchy, i.e., fast, shared local memories, that affect the performance of the overall system. When multiple jobs compete for processor time, they potentially evict each other's cache data that must be reloaded later.
The initial idea to gain more control on the caches in order to perform resource management, is to lock the content of cache lines. Thus, their content cannot be evicted by the activities of competing jobs on the same or on other cores. A more advanced idea is *cache coloring*, in which memory addresses are mapped to certain cache lines. As a result, the accesses of the individual jobs do not interfere with each other. This can either be achieved by the operating system [64] with or without support of dedicated hardware of the underlying platform, or by means of the compiler that produces the binary files that are run on the platform [65]. [66] gives a good overview of the operating system- and compiler-based techniques. An alternative approach is the use of local software controlled scratch-pads instead of hardware controlled caches, as e.g., proposed in [67].

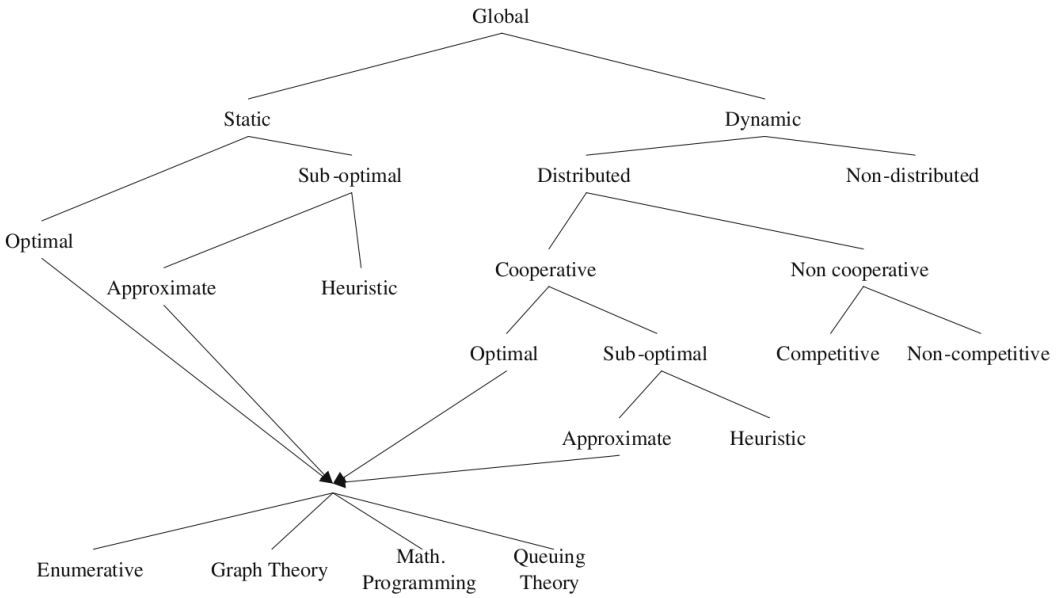


Figure 1.2: *Taxonomy of resource assignment algorithms, inspired by [2].*

Similar challenges impose other processor components, e.g., translation lookaside buffers (TLBs) that are commonly used in today's systems to speed up memory management. Further "holes" in the isolation of the processes are imposed by the management of the interconnects, e.g., buses to caches and main memories. In [68], modifications to the bus arbitration policy have been proposed to solve this issue[3].

Further challenges arise with the management of the main memory itself, e.g., unpredictable access times to SDRAM elements caused by refreshes or contention at the memory controller when multiple cores access the same memory region. To overcome these issues, predictable memory controller for domain specific DDR2 SDRAMs [69] or generic DDRx SDRAMs have been proposed [70].
In general, the management of I/O devices controlled via DMA controllers that additionally and independent of the processors trigger transactions on the interconnect imposes non-trivial challenges. Finally, system maintenance interrupts and features like hardware pre-fetching, and dynamic voltage and frequency scaling used for power save modes introduce even more challenges to real-time resource management.

1.4 Thesis Outline

The rest of this thesis is organized in the following chapters:

Chapter 2: — this chapter presents the original slot shifting algorithm [1], which is composed of two main parts: the offline phase and the online phase. First, we explain the underlying concepts of the offline phase, which resolves the complex constraints of offline tasks using an offline scheduler. The task of the offline scheduler is to create an offline scheduling table, which it then annotates with additional information about spare capacities, i.e., task flexibilities. We describe the working-principle of the online phase, which at runtime makes twofold use of the annotated offline scheduling table: First, in order to obey the offline tasks' complex constraints, and second, to integrate arriving aperiodic tasks feasibly into the schedule.

Chapter 3: — in this chapter, denominated "Online admission of non-preemptive aperiodic tasks in time-triggered schedules", we portray our approach to include the handling of non-preemptive firm aperiodic tasks into time-triggered systems based on the slot shifting algorithm [61]. Our approach is based on the slot shifting algorithm and does not require to create a new additional offline scheduling table, instead it updates an existing table and induces only minimal memory overhead. The focus of our approach is to minimize the response time of the non-preemptive tasks. We propose two different approaches to guarantee the non-preemptive task and we analyze the computational complexity of our approach.

Chapter 4: — this chapter presents our global multicore slot shifting algorithms and their rationale. Since no optimal online scheduling algorithm for aperiodic tasks

[3]Note that our measurements presented in section 5.3 are performed on the very same simulation engine, using their programmable TDMA bus arbiter to guarantee access to the bus.

exists, our algorithms aim at increasing the probability of successful acceptance of the aperiodic task at the destination core. When integration on the local core fails, the first algorithm bases its delegation, i.e., migration, decision for the aperiodic job on the available spare capacities on the other cores of the system. The second algorithm relieves the delegating core from the computationally intense process of determining a suitable destination core. Instead, the remaining cores of the system negotiate among themselves a suitable candidate.

Chapter 5: — in this chapter, we perform a variety of experiments to analyze the original as well as the global slot shifting algorithms introduced in the previous chapter. The experiments described in this chapter aim at quantifying the efficiency of the algorithms:
We utilize MPARM, a cycle-accurate MP-SoC simulation engine developed at the MicRel Lab at the University of Bologna [71], to determine the runtime overheads associated with the slot shifting algorithms. The simulations run on a simulated ARM multicore system and use a "bare-board" C-implementation of the respective slot shifting algorithms. Thus, the results of the experiment are reproducible and it is ensured that neither user nor operating system interference influence or falsify the results. Further, based on this runtime analysis, we develop, discuss, and analyze further approaches to reduce the runtime overheads.

Chapter 6: — in this chapter, we perform many experiments to elaborate on the effectiveness of the slot shifting algorithms. In order to achieve this, we simulate the slot shifting algorithms in a Linux environment on the high performance cluster Elwetritsch of the University of Kaiserslautern. In a multitude of experiments, we evaluate the influence of different parameters, e.g., varying the utilization created by aperiodic and offline tasks, the deadline of aperiodic tasks, and their arrival pattern. We analyze the resulting acceptance ratio, the number of performed acceptance tests, and the resulting response times of the aperiodic jobs and discuss our findings.

Chapter 7: — this chapter addresses the resource management on multicore platforms using slot-shifting-based algorithms in a generic adaptive resource management framework. The system design is based on and inspired by the system design of the resource manager and the underlying Linux in the ACTORS project [72, 3]. Its patched Linux kernel features real-time capabilities, enforced by a constant bandwidth server (CBS) mechanism that ensures temporal isolation between the different applications running on the same core. The system runs a resource manager, a privileged user space application that controls and manages the resource access among the running applications by means of the CBS. The focus of the beginning of this chapter is to present the system design and the interfaces. Furthermore, the chapter discusses the implementation aspects and the challenges faced to integrate slot shifting into the Linux-based ACTORS resource management framework. Then, we present the results obtained from our implementation of a "slot shifting logic" into the ACTORS resource manager. The results highlight the benefits of our approach.

Chapter 8 — in this chapter, we summarize the main contributions of this thesis. We give pointers for future directions of research and conclude the thesis with final remarks.

Chapter 2

Distributed Time-Triggered Systems and Event-Triggered Activities

The slot shifting algorithm is a real-time scheduling algorithm that combines the benefits of both event- and time-triggered scheduling. Previously, in chapter 1, we gave a detailed overview of the properties of event-triggered and time-triggered scheduling. In this chapter, we describe underlying assumptions, working principles, and properties of the slot shifting algorithm, whose aim is manifold. First, it provides predictability and deterministic runtime guarantees for the execution of pre-planned tasks, by means of an offline scheduling table. Second, while ensuring these properties, slot shifting provides flexibility to enable the system to react to unforeseen events. To achieve this runtime behavior, slot shifting aims to guarantee the timely execution of aperiodic tasks. Whether slot shifting can provide this guarantee or not is determined by an online acceptance test, which itself is based on the information found in the offline scheduling table. The integration of aperiodic tasks into the schedule at runtime must be performed such that the schedulability of already guaranteed tasks is not harmed and such that all real-time constraints of the tasks are fulfilled. Finally, while providing all the aforementioned properties, slot shifting aims to minimize the response time of the aperiodic tasks.

This chapter explains the original slot shifting algorithm in detail and is structured as follows. First, we explain the underlying assumptions that form the foundation of the slot shifting algorithm. Then, we describe the principles and the system model used by the algorithm. This is followed by a general discussion of the underlying task model and an explanation how task dependencies are modeled. The slot shifting algorithm itself is composed of two phases, each of which is explained in a separate section of this chapter: First, the *offline phase* which resolves the constraints of the offline tasks and creates the offline scheduling table. Additionally, the offline phase annotates this table with information about the flexibility of the offline tasks to ease the integration of aperiodic tasks into the schedule at runtime. Second, the *online phase* during which tasks are executed according to the offline scheduling table in EDF-fashion. When firm aperiodic tasks arrive, slot shifting tries to dynamically integrate them into the schedule at runtime; soft aperiodic tasks are scheduled when the processor idles.

2.1 Introduction

Slot shifting [1] is a real-time scheduling algorithm for distributed systems. Such systems are composed of multiple communication nodes and processing nodes linked together via a network interconnect. The logical separation into processing and communication nodes allows to split task and network scheduling. The work discussed in this thesis does not cover the network scheduling of the communication nodes. Throughout the rest of this thesis, we will use the terms processor, and core interchangeably, when we refer to these (processing) nodes.

Slot shifting uses a discrete time model [73]. Underlying this discrete time model is the notion of a measurable event, which we define as proposed by Kopetz:

Definition "An *event* is an occurrence at a point in time, i.e., a happening at a cut of the time-line, which itself does not take any time" [74].

For all nodes of a distributed system, slot shifting assumes a global time, whose progression is triggered by equidistant events, detected by an independent external observer. This external observer counts the occurrence of these events starting from zero to infinity. The granularity of this globally synchronized time model, i.e., the time interval that separates two of these events, is called *slot*. More formally, the slot i is defined as the interval $[event_i, event_{i+1}]$.

Figure 2.1 details our slot definition. A slot is sub divided into two time intervals, because any invocation of a real-world scheduler takes some non-zero time. In the first time interval Δt_S that starts at the beginning of the slot, the scheduler is invoked to make a decision which job of which task to schedule next. After the scheduling decision has been made, the selected job is scheduled within the second time interval Δt_E, i.e., for the remaining time of the slot. For the rest of this thesis, we assume that Δt_S is much smaller than Δt_E.

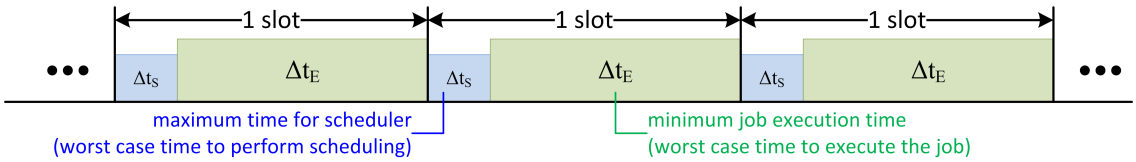


Figure 2.1: *Definition of a slot.*

The workload, i.e., all activities that are to be performed by the processor, is represented by tasks featuring complex constraints. The jobs of these tasks are fully preemptive and communicate with each other by reading data at the beginning of their execution and writing results at the end of their execution. The worst case execution time[1], C, of all jobs is bounded and known before runtime of the system.
Slot shifting assumes that all these tasks can be split into two groups: The first group consists of activities that are known before the runtime of the system and thus can be

[1] Or as in [1]: maximum execution time, MAXT.

pre-planed. The second group consists of activities that need to be performed at runtime of the system. Their arrival patterns are completely unknown, i.e., they can arrive on any processor at any moment in time. The former group consists of the so-called *offline tasks*; whereas the latter group, is comprised of so-called *online tasks*.

Offline tasks are entities of work which feature predictable, pre-planned job arrival times. Slot shifting supports precedence constrained offline tasks, i.e., the start of a job of one task depends on the completion of a job of another task. The basic assumptions for these tasks are similar, but not identical to these of the Liu and Layland task model explained in section 1.3.1. Although the tasks feature a bounded execution time, they are not independent and are not explicitly periodic[2]. Further, not all offline tasks per se feature a deadline constraint. The offline phase of slot shifting later assigns deadlines to all jobs of offline tasks to enforce the precedence constraints. The last assumption from Liu and Layland is that all overheads related to scheduling of the tasks are negligible. In the course of this thesis, we will analyze to which extend this assumption holds for real implementations of the slot shifting algorithm. We discuss the results of our findings in chapter 5. There, we provide detailed results that we obtained from a variety of experiments to evaluate the different versions of the slot shifting algorithm.

Online tasks are independent entities of work and their jobs arrive without predetermined arrival pattern. They are modeled as *soft* or *firm*, depending on whether they feature deadlines that must be obeyed or not.
The worst case execution time of jobs of both offline and online tasks are assumed to be known at runtime of the system.

As already mentioned, the offline tasks are subject to precedence constraints. We express such constraints in a *precedence graph* (PG). A PG is a directed acyclic graph in which nodes represent the tasks and edges represent precedence constraints. The complete job set consists of jobs of offline tasks, represented by one or multiple PGs. To model periodic behavior of the tasks, the PGs are periodic with a fixed time interval between two successive activations. Thus, every PG implicitly has a deadline, i.e., a time by which the execution of all jobs of the tasks that constitute the PG must have finished.

Tasks in a PG which have no immediate predecessor task are called *entry tasks*. Similarly, tasks in a PG which do not have any successor task are called *exit tasks*. All tasks of a PG, except the exit tasks, are only constrained by their worst case execution time and by precedence relations expressed by the edges of the PG. Exit tasks implicitly feature a deadline constraint caused by the deadline of the PG. Figure 2.2 shows an example PG consisting of three tasks τ_A, τ_B, and τ_C. The deadline of the PG is set to the beginning of the 25th slot and the worst case execution times of the individual tasks are listed in the right part of the figure. Task τ_A is the only entry task of this PG and the task τ_C is the only exit task. In this example, the task τ_B depends on task τ_A and thus can only be scheduled after τ_A has finished its execution. Similarly, the task τ_C

[2]Nevertheless, periodic behavior is achieved by means of the periodicity of the precedence graphs, as explained later.

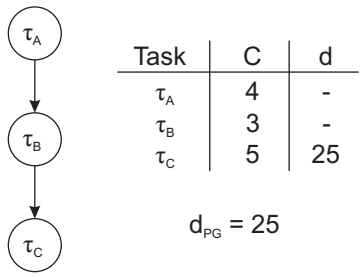


Figure 2.2: *Example precedence graph (PG) consisting of three tasks.*

depends on τ_B and hence can only be scheduled after τ_B has completed. Implicitly, the exit task features a deadline equal to the deadline of the PG, i.e., $D_{PG} = D_C = 25$.

2.2 Offline Phase

2.2.1 Overview

In the first phase of the slot shifting algorithm, the offline phase, the complex precedence constraints of the tasks specified by the PGs are resolved by means of an offline scheduler. This offline scheduler maps the individual jobs of the tasks to the nodes and determines the order of execution, taking the precedence constraints into account. It calculates the earliest start times and the deadlines of these jobs based on the deadline of the PGs, the worst case execution times of the jobs, and the sending and receiving times of inter-processor messages.

The aim of slot shifting is to determine the available leeway in the schedule in the offline phase and to provide this flexibility later in the online phase to react to aperiodic jobs at runtime of the system. The offline scheduler calculates the earliest start times and the latest possible deadlines of the offline jobs and maps them to the processors. Where possible, jobs mapped to the same processor and running consecutively are combined to so called *scheduling blocks*. As all dependencies have already been resolved by the offline scheduler, scheduling blocks are treated as independent jobs[3], for more details see [75].

Figure 2.3 shows an example with two PGs that altogether consist of six tasks. The top left of the figure shows the two PGs and the top right of the figure lists the worst case execution times of the tasks and the deadlines of the PGs. The bottom part of the figure shows a schedule which results from mapping the jobs of the tasks of the two PGs to two processors; the two arrows indicate inter-processor messages.

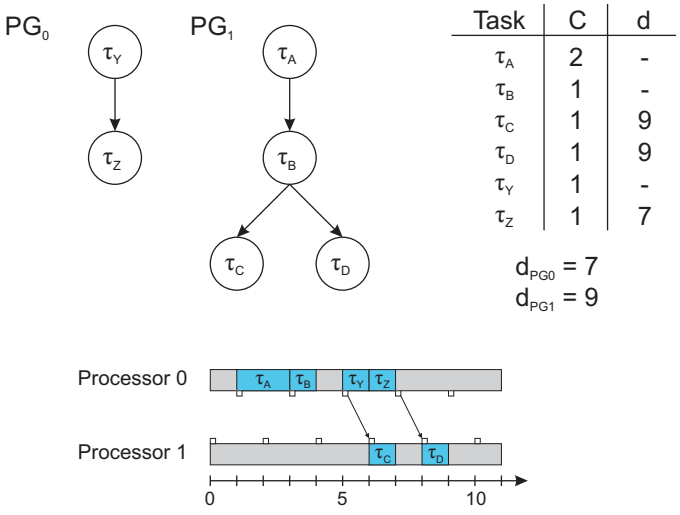
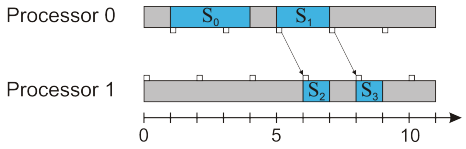


Figure 2.3: *Example PGs and schedule, inspired by [1].*

Figure 2.4 shows the schedule of the four resulting scheduling blocks. On the bottom of the figure, there is a table that lists the earliest start times (est), the worst case execution



Scheduling Block	est	C	d
S_0 (τ_A, τ_B)	0	3	5
S_1 (τ_Y, τ_Z)	0	2	1
S_2 (τ_C)	6	1	8
S_3 (τ_D)	8	1	9

Figure 2.4: *Schedule for example with scheduling blocks, inspired by [1].*

times (C), and absolute deadlines (d) of the scheduling blocks. The earliest start times and the deadlines of tasks determine the *execution windows*[4] of the individual jobs. This information together with general job information (job ID, worst case execution time, message sending and receiving times) is stored in an offline scheduling table.

In the next step, this offline table is annotated with additional information about the flexibility of the jobs inherent to the given scheduling table. This procedure is divided into the following steps: First, the algorithm calculates *capacity intervals*, i.e., sets of consecutive slots that cover disjoint intervals of time. Then, it assigns all jobs to these intervals. Finally, the algorithm calculates for every interval based on the properties of their constituting jobs the amount of available free slots, the *spare capacity*. The purpose of the annotation is to ease scheduling decisions during the succeeding online phase. Note that the actual resulting execution windows of the jobs during the online phase are not necessarily identical to the offline calculated capacity intervals. In other words, the capacity intervals serve to calculate the spare capacity values. They do not restrict the flexibility of the scheduler at runtime; jobs are ready to run and will potentially be scheduled at runtime prior to the capacity interval they have been assigned to.

2.2.2 Calculation of Intervals and Spare Capacities

In general, capacity intervals[5] are determined by the deadlines of the offline jobs. Thus, the specification of these intervals first requires the ordering of all offline jobs according to their deadlines. For every distinct deadline value of a job from the job set there must exist a corresponding capacity interval. Jobs with identical deadlines belong to the same interval.

To determine the start of a capacity interval I_j, first the *earliest start time of the capacity interval*, $est(I_j)$, is defined as the minimum of the earliest start times of all jobs that belong to the interval under consideration (see Equation 2.1). Based on the

[3] From now on, we will not distinguish between jobs and scheduling blocks.

[4] Sometimes they are also called execution intervals.

[5] Note that throughout this thesis the terms *interval* and *capacity interval* are used interchangeably. To avoid confusion with the term execution interval, we will from now on use the terms execution windows and (capacity) intervals.

earliest start time of the interval, the start of the interval I_j, $start(I_j)$, is defined as the maximum of the earliest start time of the interval and the end of the previous interval (see Equation 2.2).

$$est(I_j) \;=\; est(T_x) \wedge (est(T_x) = min(est(T_i)))\ \forall T_i \in I_j \tag{2.1}$$

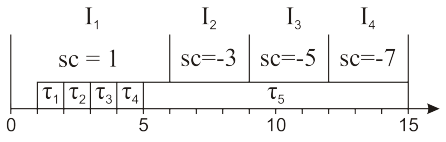
$$start(I_j) \;=\; max(end(I_{j-1}), est(I_j)) \tag{2.2}$$

Note that by this definition, two capacity intervals can be separated by at most one single empty interval, i.e., an interval without any jobs assigned to it. During an empty interval there exists no offline job that is ready to execute. Nevertheless, an empty interval can be subject to borrowing caused by its successor interval.

To define the *spare capacity* of an interval, we first define the length of an interval I_j as the difference between its end and its start: $|I_j| = end(I_j) - start(I_j)$. The spare capacity of the interval I_j is defined as the length of this interval decreased by the sum of the worst case execution times of all of the jobs of this interval and decreased by the amount of slots that are "borrowed" by the succeeding interval (see Equation 2.3). The spare capacities of the individual intervals are calculated starting from the last to the first interval in time. A resulting positive spare capacity of an interval expresses the "the amount of unused resources and leeway" [1] which will later be available during the online phase for the execution of jobs of aperiodic tasks. A resulting negative spare capacity of interval I_j expresses the number of slots that the capacity interval I_j "borrows" from the preceding interval I_{j-1} to accommodate the jobs that belong to I_j. Or in other words: a negative spare capacity value of a capacity interval indicates that the sum of the worst case execution time of all jobs in the successor interval is greater than the length of this successor interval.

$$sc(I_j) \;=\; |I_j| - \sum_{T_i \in I_j} C_i + min(sc(I_{j+1}), 0) \tag{2.3}$$

As the jobs are not allowed to execute after their interval end (since this would lead to a deadline miss), this automatically creates "pressure" on the preceding interval(s). This pressure is represented in the model by reduced spare capacities, which is reflected by the rightmost term in Equation (2.3). Thus, the successor interval "borrows" slots from its predecessor interval to accommodate all jobs that the successor interval hosts. Note that borrowing is not necessary limited to a single predecessor interval but it can involve multiple intervals. In fact, an interval hosting multiple or a very long job could cause a chain of intervals borrowing from their immediate predecessor intervals. An example of such a "chain of borrowing" is depicted in Figure 2.5. At the right side of the figure, a



Task	est	C	d	Interval
τ_1	0	1	6	I_1
τ_2	0	1	9	I_2
τ_3	0	1	12	I_3
τ_4	0	1	15	I_4
τ_5	0	9	15	I_4

Figure 2.5: *Example job set showing a chain of borrowing.*

table lists the properties of the individual jobs. The jobs are assigned to the intervals according to their deadlines. The resulting capacity intervals with their spare capacities are listed in the schedule on the left side of the figure. Notice that, since job τ_4 and τ_5 both feature a deadline of 15, both belong to interval I_4. The worst case execution time of τ_5 does not completely fit into I_4, thus I_4 borrows 7 slots from the preceding interval I_3, which in turn borrows 5 slots from I_2, which also borrows 3 slots from I_1.

If not mentioned otherwise, we assume for the offline scheduling table in the rest of the thesis that job IDs are system-wide unique and interval IDs are unique per processor. Further, job IDs are assigned in the same order as the jobs are scheduled in the scheduling table: All jobs are numbered in the order of their appearance, i.e., depending on their earliest start time and deadline.
We further assume that the offline scheduler assigns the interval IDs in strictly monotonically increasing fashion, starting from 1. Empty capacity intervals are treated like capacity intervals that host jobs. If on a processor the first job does not start at time zero, an additional empty capacity interval is added in front for completeness of the scheduling table. In case of gaps between the intervals, empty intervals are inserted between the already existing intervals. Finally, an additional empty interval is inserted after the deadline of the last job to equalize the lengths of the scheduling tables on all processors.

2.3 Online Phase

During the online phase of slot shifting, the scheduler on each processing node maintains a list of ready jobs based on information found in the offline scheduling table. At the beginning of each slot, the scheduler first checks whether aperiodic jobs have arrived during the last slot. As long as no aperiodic job has arrived, the scheduler selects from the ready list a job for execution according to the EDF scheduling algorithm. Soft aperiodic jobs require no special treatment (see next section). For each newly arrived firm aperiodic job, denoted as $\tau_{aperiodic}$, the slot shifting algorithm performs an *acceptance test*. We describe the working principle of the acceptance test in detail in the following section. If the acceptance test fails, then the aperiodic job is rejected by the system. Otherwise, the job is integrated into the schedule by the so called *guarantee algorithm*. The exact steps of the guarantee algorithm are described in section 2.3.2. In section 2.3.3, we introduce the Shorten Deadline algorithm, an improved version of the guarantee algorithm. After that, we describe the decision making process of the slot shifting scheduler and the mechanism that is required to update the spare capacity values of intervals when jobs execute prior to their assigned interval, in section 2.3.4. Finally, we exemplify the working principles of slot shifting with two examples in section 2.3.5.

2.3.1 Acceptance Test

For each newly arrived firm aperiodic job, $\tau_{aperiodic}$, the slot shifting algorithm performs an acceptance test based on the available spare capacities in the capacity intervals up to the deadline of the job. In the case that multiple firm aperiodic jobs have arrived during

the last slot, the acceptance test is performed for one job after another on an earliest deadline first basis. After each successful acceptance, the guarantee algorithm is invoked for the corresponding aperiodic job; while jobs for which the acceptance test failed are rejected by the algorithm, since the system features not enough available resources to ensure timely execution. For soft aperiodic jobs, the acceptance test is not invoked. Instead, they are added to a ready queue of their own and are run as background jobs, more precisely, whenever positive spare capacity is available and no firm aperiodic job is ready. As soft aperiodic jobs are only run under this condition, they cannot harm other jobs in the system. Since soft aperiodic jobs themselves feature no deadline, their potential delay due to other jobs will not cause any issues for them, except an increased response time.

The purpose of the acceptance test for firm aperiodic jobs is to determine whether there are enough computational resources available to safely execute them without harming other jobs in the system. As the very first step, the acceptance test checks for each aperiodic job $\tau_{aperiodic}$ the trivial condition $d_{aperiodic} \geq t_{now} + C_{aperiodic}$. After that, the actual acceptance test is performed. For each $\tau_{aperiodic}$, there are three different parts of spare capacities considered for the test:

- The remaining spare capacity in the current interval of time, $sc(I_c)$. Note that the spare capacity in the current interval is always non-negative.
- The spare capacity of all complete intervals between the current interval in time I_c and the deadline of the aperiodic job $D_{aperiodic}$. I_l denotes the last of these complete intervals.

$$\sum_{\substack{c<i\leq l \\ end(I_l) \leq d_{aperiodic} \wedge \\ end(I_{l+1}) > d_{aperiodic}}} max(0, sc(I_i)) \tag{2.4}$$

- If the deadline of the aperiodic job does not coincide with the end of an interval, we need to add a fraction of the spare capacity of the last interval, i.e., the available spare capacity up to the deadline of the aperiodic job:

$$max(0, min(sc(I_{l+1}), d_{aperiodic} - start(I_{l+1}))) \tag{2.5}$$

If the sum of these three spare capacities is less than the worst case execution time of the aperiodic job, then $\tau_{aperiodic}$ is rejected by the system as sufficient guarantees cannot be provided for its execution. Otherwise, $\tau_{aperiodic}$ is integrated into the offline scheduling table by means of the guarantee algorithm. The following section describes the exact steps that the guarantee algorithm performs.

2.3.2 Guarantee Algorithm

As the first step, the guarantee algorithm tries to directly add the new aperiodic job $\tau_{aperiodic}$ to an already existing capacity interval. This can only be realized if the deadline of $\tau_{aperiodic}$ coincides with the end of an existing interval I_l. In that case, the guarantee

algorithm adds the aperiodic job $\tau_{aperiodic}$ to I_l. To reflect that a new job has been added to the interval, the guarantee algorithm reduces the spare capacity of I_l by the worst case execution time of the new job, $C_{aperiodic}$. If the spare capacity of the interval becomes (or is already) negative, the spare capacity of the previous interval must be updated, too. Note that this updating might affect multiple predecessor intervals that form a "chain of borrowing".

Otherwise, if the deadline of the aperiodic job does not coincide with the end of any existing interval, the guarantee algorithm splits an existing interval. The algorithm selects the interval I_{l+1} in which the deadline of the arrived aperiodic job lies and cuts it into two disjoint intervals. The first new interval I_{l1} starts exactly when the original interval I_{l+1} started and ends with the deadline of the aperiodic job. The second new interval I_{l2} starts at the end of the first interval and ends exactly as the original interval I_{l+1}. All jobs that originally belonged to I_{l+1} are reassigned to the new interval I_{l2}. In the next step, the guarantee algorithm adds the aperiodic job to interval I_{l1}. The final step of the guarantee algorithm is to update the spare capacities to reserve the slots needed for the execution of the newly added job. To achieve this, it is necessary to recalculate the spare capacities starting from the new interval I_{l2}, going backwards in time. Note that a chain of borrowing might start in interval I_{l1} or I_{l2}. If that is the case, all spare capacity values need to be recalculated, until an interval with non-negative spare capacity is reached.

Independent whether the guarantee algorithm only added the job or had to split an existing interval, in the next step, slot shifting adds the new aperiodic job to the list of ready jobs.

2.3.3 Shorten Deadline Algorithm

For performance reasons, we modified the original slot shifting algorithm, more exactly the acceptance test and the guarantee algorithm. This improved slot shifting algorithm shortens the deadlines of aperiodic jobs as much as possible such that they coincide with the ends of already existing intervals. We refer to this new feature of the slot shifting algorithm as *SDL* (**S**horten **D**ead**L**ine algorithm). Listing 2.1 presents the simplified pseudocode of the SDL algorithm.

In essence, SDL aims at improving the runtime requirement in slots in which aperiodic jobs arrive: SDL tries to avoid costly interval splits and thus successive updates of their spare capacities. Hence, the time required to online integrate aperiodic jobs into the scheduling table will reduce.

Listing 2.1: *Pseudocode of Shorten DeadLine (SDL) algorithm.*

```
bool Algorithm_SDL(tTask* task_ptr, int current_slot, tInterval* current_interval_ptr)

int sum_sc = 0;
int dl = task_ptr->dl;

if(dl < current_slot + wcet)
        return FALSE;


// then calculate sc up to dl of aperiodic task
for(struct tInterval* i = current_interval_ptr; dl > i->start; i = i->next)
{

        // dl coincides with interval->end OR dl > interval->end
        if(dl >= i->end)
        {
                sum_sc += max(0, i->spare_capacity);
        }

        // dl is inside the interval
        else {
                if(i == current_interval_ptr)
                        sum_sc += min( max(0, i->spare_capacity), dl - current_slot );
                else
                        sum_sc += min( max(0, i->spare_capacity), dl - i->start );
        }

        // if sufficient spare capacity, perform SDL
        if(sum_sc >= wcet)
        {
                if(dl > i->end)
                {
                        // dl in future and enough sc already reached, so shorten dl
                        task_ptr->deadline = i->end;
                }
                return TRUE;
        }

} //end of for loop.

return FALSE;
```

2.3.4 Scheduling and Maintenance of Spare Capacities

The last but most important decision that the slot shifting algorithm has to make is to decide which job to schedule next. We identify the following three cases:

1. There is no job ready for execution, thus the scheduler cannot select any job to run, the processor is left idle and the spare capacity of the current interval is decreased by one.

2. There is at least one job ready to execute and the spare capacity value in the current interval is exactly zero. In this case, the scheduler has to select an offline or an online guaranteed job to execute. Any other decision of the scheduler would lead to a deadline miss of a guaranteed job. The scheduler picks the guaranteed job with the earliest deadline for execution; ties are broken arbitrarily.

3. There is at least one job ready to execute and the spare capacity value in the current interval is positive. Thus, the scheduler selects a soft aperiodic job to execute and reduces the available spare capacity by one slot. Note that by applying this rule, soft aperiodic jobs are preferred to already guarantee aperiodic jobs, i.e., the responsiveness of soft aperiodic jobs is improved.
   The situation might arise that there is no soft aperiodic job ready for execution. In this case, the scheduler schedules the guaranteed job with the earliest deadline, if there is one; otherwise the processor is left idle and the spare capacity of the current interval is decreased (as in case 1).

The execution of an offline or online guaranteed job is already reflected by the table and thus, the spare capacity value usually does not change. An important exception to this rule occurs however, if a guaranteed job τ_g executes prior to its assigned interval. In this case, the spare capacities of potentially multiple intervals need to be maintained: As the scheduler selected τ_g to execute earlier than planned in the table, this results in one more slot being available for future aperiodic jobs in the interval the selected job τ_g is assigned to. To reflect this, the available spare capacity of the interval τ_g belongs to is increased by one slot. If this interval borrows slots from a previous interval, or multiple previous intervals (chain of borrowing), the spare capacity of the affected interval(s) must be increased by one slot, too.

After updating the interval that hosts τ_g and updating all intervals affected by borrowing, also the current capacity interval needs to be updated. Since one slot of computation time is spent to execute a job that was originally supposed to run later, the available slots for aperiodic execution reduce by one slot. In other words, the available spare capacity of the current interval needs to be decreased by one. The only exception to that rule occurs, when the early starting job was hosted in an interval that borrowed slots from the previous interval(s) and this chain of borrowing affects the current capacity interval. As the table already reflects this borrowing in this case, the current intervals spare capacity remains unchanged. Nevertheless, the spare capacity values of the other intervals need updating.

Note that during the runtime of the online phase, the spare capacity value of the current interval can never become negative as this would indicate an unavoidable deadline miss of one of the jobs of the job set.

2.3.5 Example Schedules

This section demonstrates with two examples the working principles of slot shifting.

2.3.5.1 Example 1

The first example schedule shows normal job execution, the early start of an offline job, the integration of a firm aperiodic job and the spare capacity update mechanism. The parameters of the job set are listed in Table 2.1. Figure 2.6 shows the resulting

Job	est	C	d	Interval
τ_1	0	1	4	1
τ_2	0	1	10	2
$\tau_{aperiodic}$	3	4	7	–

Table 2.1: *Parameters of the job set for example 1.*

schedule. The figure is divided into six sub figures that represent intermediate steps of the schedule. The topmost part presents the initial intervals (and their jobs) and their calculated spare capacities.

At $t_{current} = 0$, the scheduler selects the job with the earliest deadline, τ_1, to run. Figure 2.6b shows the situation afterwards: τ_1 has been executed and all spare capacity values remain unchanged.

Then, the scheduler selects τ_2 to run; the updated spare capacities are shown in Figure 2.6c. Job τ_2 runs outside its assigned interval, thus the maintenance mechanism of slot shifting increases the available spare capacity of its assigned interval I_2 by one slot. Note that running τ_2 in I_1 reduces the available spare capacity in I_1 by one slot.

At the beginning of the next slot, we see that there is no job ready for execution, thus the processor remains idle and one slot of spare capacity of interval I_1 is lost. During this idle slot, a firm aperiodic job $\tau_{aperiodic}$ arrives. Figure 2.6d shows the spare capacity values just before the acceptance test is invoked. The acceptance test sums up the available spare capacity from $t_{current}$ until the $d_{aperiodic}$ and checks whether $\tau_{aperiodic}$ can be accepted. After the successful test, $\tau_{aperiodic}$ is integrated into the schedule. The guarantee algorithm adds this job by splitting I_2 into I_2 and I_3 and updates the spare capacities: The spare capacity of I_3 changes to 3, the spare capacity of I_2 changes to -1 and the spare capacity of I_1 changes to 0. Figure 2.6e shows the spare capacities in the moment of time directly after the aperiodic job has been integrated.

At the beginning of the next slot, the scheduler selects $\tau_{aperiodic}$ to run for one slot. Because one slot of a job that belongs to I_2 was executed prior its interval, i.e., in interval I_1, the spare capacity of I_3 is incremented by one slot. Since I_2 previously borrowed

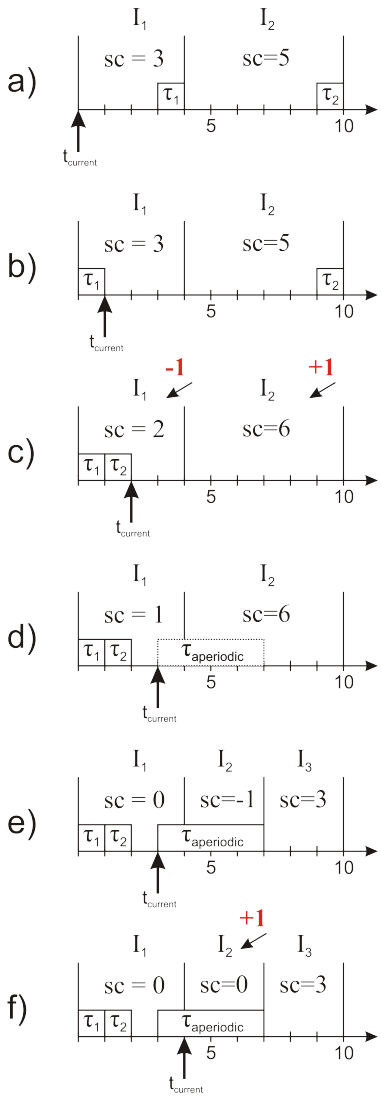


Figure 2.6: *Small example schedule showing early start of a job, integration of a firm aperiodic job and spare capacity update mechanism.*

one slot from its predecessor interval I_1, the update mechanism would normally increment the previous interval's spare capacity also by one slot. However, as the previous interval is the interval in which $\tau_{aperiodic}$ was executed (and which thus lost one slot of spare capacity), the spare capacity value of I_1 remains constant. Figure 2.6f shows the resulting spare capacities.

2.3.5.2 Example 2

Figure 2.7 explains the maintenance mechanism for spare capacities using the same job set and schedule as presented in section 2.2.2 in Figure 2.5. The figure is divided into seven sub figures (a-g), each showing the changes of the spare capacity values as time progresses slot-by-slot.

Figure 2.7a shows the intervals with their initial, unmodified spare capacity values of the offline scheduling table.

At $t_{current} = 0$, the scheduler selects τ_1 to run, as it features the earliest deadline. Since τ_1 executes inside its interval, the spare capacities remain unchanged, as can be seen in Figure 2.7b.

At the beginning of the next slot, the scheduler selects τ_2 to run. This job had been offline assigned to interval I_2, thus there is the need to update of the spare capacity of interval I_2. Early execution of offline or online guaranteed jobs usually decreases the available spare capacity in the current interval by one slot. In this example however, borrowing from I_2 into the current interval took already place before: Since the sum of execution times of the jobs assigned to I_2 is more than length of the interval, I_2 has to borrow one slot from interval I_1. The single slot of time that is spent to execute τ_2 prior its interval I_2 is gained again since after the execution I_2 does not borrow any slots I_1 anymore. In other words, the early execution of τ_2 is already reflected by the spare capacity values in the table, hence the spare capacity of the current interval remains constant, see Figure 2.7c.

The next job that the scheduler selects is τ_3. Again, the update mechanism is triggered, since this job starts prior its interval I_3. The intervals I_3, I_2, and I_1 form a chain of borrowing, thus causing a ripple-effect of spare capacity updates. Figure 2.7d shows the spare capacities after the update has been performed. Notice that the spare capacity of I_1 remains constant, although I_1 is part of the borrowing chain. The reason is the same as one time slot earlier when τ_2 was executed: The spare capacity value of the current interval I_1 already reflects the early start of τ_3.

One slot later, at $t_{current} = 3$ the scheduler selects τ_4. Since τ_4 belongs to I_4, which is part of a chain of borrowing, the update mechanism increments the spare capacity value of the affected intervals. The spare capacity value of I_1 again remains unchanged as the interval "lost" one slot to execute τ_4 prior its interval, but also gained one slot that was previously borrowed by I_2, see Figure 2.7e.

At $t_{current} = 4$, τ_5 is scheduled and because this job also belongs to I_4, the same updates as one slot before take place, see Figure 2.7f.

Finally, Figure 2.7g shows the spare capacities after τ_5 has been scheduled for one more slot. The spare capacities of the intervals I_4, I_3, and I_2 are incremented by one

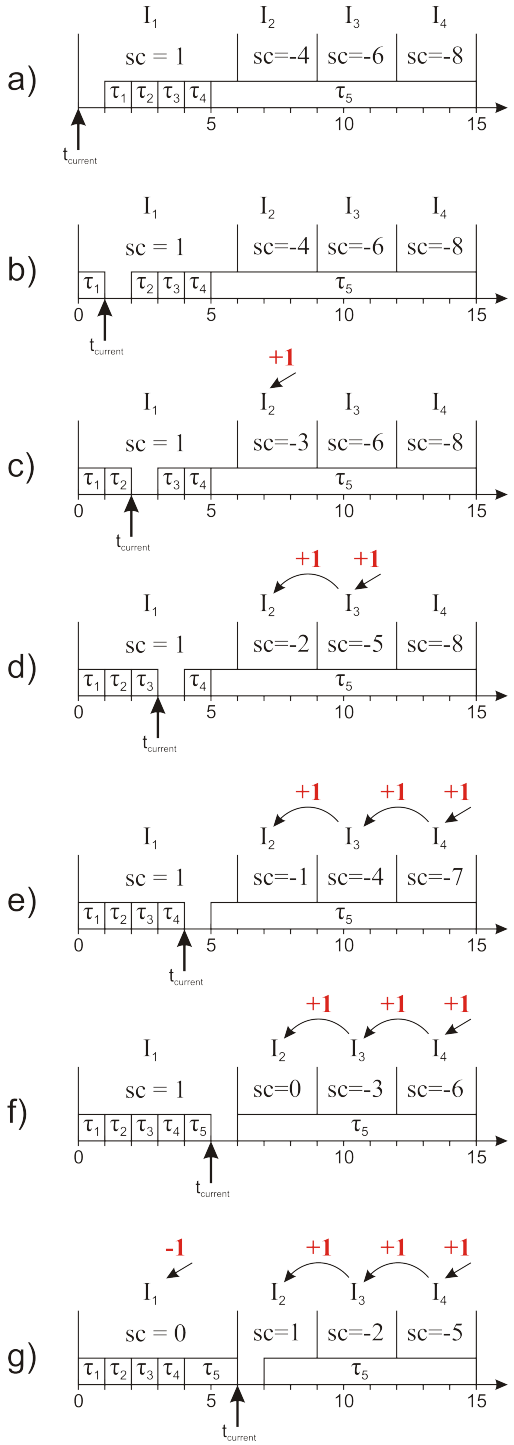


Figure 2.7: *More complex schedule involving multiple spare capacity updates.*

slot. Notice that this time, the spare capacity value of I_1 decreases by one slot to zero. The reason is that one slot of execution time was spent for τ_5. This single slot was not compensated for in the table and not by freeing one previously borrowed slot; no borrowing took place before, which can be seen in Figure 2.7f. In the previous slots, the neighboring interval had always previously borrowed at least one slot from the current interval which then was freed and thus kept the spare capacity of the current interval constant.

Chapter 3

Online Admission of Non-Preemptive Aperiodic Tasks in Time-Triggered Schedules

In this chapter, we describe our approach to add support for non-preemptive tasks to the slot shifting algorithm. The chapter starts with a brief introduction and continues with a description of the main challenges. Then, the following section describes the methodology we use. It starts with the underlying assumptions concerning the offline scheduling table and initial constraints for the task parameters. We then describe a simple approach to tackle the problem with the given constraints. There, we focus on the required changes for the online acceptance test. Then, we weaken the constraints and describe our final approach to integrate non-preemptive tasks into slot shifting, as described in [61]. It includes a detailed description of the modified acceptance test and the runtime mechanisms to ensure non-preemptive execution of the tasks. After that follows a section explaining the final approach by means of an example schedule. Finally, we conclude this chapter with a brief discussion of the properties of our approach.

3.1 Introduction

In general, non-preemptive tasks are required to support time-sensitive parts of code. They are used to, e.g., perform interactions with hardware with stringent timing requirements. Using non-preemptive tasks, a system can avoid perturbations induced by interrupt service routines or higher priority tasks. Further, the interferences caused by operating system and scheduler routines can be decreased. In systems without support for synchronization primitives, non-preemptive tasks are used to provide secure data access to shared resources of the system.

In the previous chapter, we discussed the slot shifting algorithm, which is divided into an offline phase for complexity reduction and creation of an offline scheduling table and in an online phase for scheduling the offline tasks and integration of aperiodic tasks at runtime. The slot shifting algorithm is a preemptive scheduling algorithm: Every slot, the online scheduler can decide to select a different task. Non-preemptive tasks are not supported and thus, their execution leads to irregular system behavior. In this chapter, we present an extension to the original slot shifting algorithm to support the admission of non-preemptive aperiodic tasks. The basic idea, presented in [61], is to accommodate a non-preemptive task by shifting the other tasks without harming the already guaranteed tasks.

3.1.1 Challenges

The variety of advantages listed in the previous sub section make it appealing to support non-preemptive tasks. However, challenges arise when trying to add non-preemptive aperiodic task handling on top of the existing preemptive slot-shifting-based scheduling approach:

First, the presence of non-preemptive tasks inevitably leads to a degradation of the response time of other aperiodic tasks and to non-optimality in terms of acceptance ratio of aperiodic tasks. These are unavoidable consequences of non-preemptive scheduling, since for the time the non-preemptive task executes other tasks suffer delays. As firm aperiodic tasks are subject to stringent timing constraints, they miss their deadline in the worst case due to such delay.

Second, integrating support for non-preemptive tasks requires major changes to the runtime mechanisms. Noticeably, to add a non-preemptive task to the online phase of slot shifting, the acceptance test needs to be changed. It is the purpose of the standard acceptance test to check for sufficient available spare capacities in a given interval of time. To handle non-preemptive tasks, the additional constraint applies to find sufficient spare capacities located in neighboring, consecutive slots. This new constraint must be enforced without violating the constraints of other tasks in the system. Especially the fact that some offline tasks can execute before the start of their interval complicates the problem. Further complications arise from the fact that some tasks cannot be scheduled freely within their interval, i.e., their interval is not equal to their scheduling window. Additionally, the runtime mechanisms must be changed such that they provide support

to execute these tasks in a non-preemptive fashion. This requires changes to update and maintenance functions to reflect these possibly long non-preemptive execution intervals.

Third, there exists a trade-off between minimizing the response time of the non-preemptive aperiodic tasks and the other tasks in the system. We consider this a design decision of the system engineer and we will discuss this issue in section 3.2.2.3.

3.2 Methodology

In this section, we derive the methodology to support the handling of non-preemptive firm aperiodic tasks with the slot shifting algorithm. More precisely, our method handles individual jobs of non-preemptive tasks. Our approach assumes a given offline scheduling table featuring the basic properties as described in section 2.2. We allow multiple offline jobs per interval, but we assume that at runtime all offline jobs can be scheduled freely within the capacity interval determined by their deadline. Further, we assume that the offline table has been constructed such that all jobs are placed and completely fit inside their intervals.

This has two implications: First, all jobs must feature earliest starting times smaller or equal to the start of the interval they have been assigned to. Otherwise, jobs cannot be shifted freely inside their intervals. Second, to schedule all jobs freely within their interval, intervals must be long enough to offer enough space to host all their jobs. In other words, offline jobs with distinct deadlines cannot be placed too densely as otherwise two timely close jobs could enforce interval start and end times that violate the above assumption, i.e., the interval could become too small to host its jobs. Another consequence of the above assumption is that borrowing of slots from previous intervals is not allowed.

We start with describing a simple approach that relies on the aforementioned assumptions. After that, we present the final approach to non-preemptive slot shifting that allows for borrowing of slots from previous intervals and for arbitrary earliest start times of jobs. The final approach removes the assumption that jobs must be able to freely execute within their interval and that borrowing does not take place.

3.2.1 Simple Approach

Adding jobs of non-preemptive firm aperiodic tasks to the schedule must not have any impact on the schedulability of all other already guaranteed jobs in the system. The acceptance test and the guarantee algorithm must be performed such that the timing constraints of all other jobs in the system are not violated.

The assumption that offline jobs can be scheduled freely within their capacity interval by the online scheduler implies that no borrowing of slots from other intervals takes place. In other words, the spare capacity of all intervals is non-negative. For this approach, we assume a given offline scheduling table created by the offline phase that additionally fulfills the aforementioned assumptions.

In the online phase, the offline scheduled jobs as well as the arriving preemptive aperiodic jobs are handled as described before in section 2.3. Here in this section, we

focus on a simple acceptance test for the non-preemptive jobs. A detailed discussion on how to ensure non-preemptive execution will follow in section 3.2.2.

The acceptance test for a non-preemptive firm aperiodic job τ_{NP} works as follows: Starting from the current interval in time up to the interval hosting τ_{NP}'s deadline, the algorithm iterates through all intervals. In each step, let the interval under consideration be named I_c. The acceptance test checks for available spare capacities in I_c and, depending on the content of the scheduling table, in the intervals I_{c-1} and I_{c-2}. Let I_p denote the interval of time created by uniting all intervals currently checked by the acceptance test. We identify the following cases, also shown in Figure 3.1:

Case 1: $I_p = I_c$
There is either no predecessor interval I_{c-1}, or the spare capacity of I_{c-1} is zero. Thus, only the spare capacity of I_c is considered.

Case 2: $I_p = I_c \cup I_{c-1}$
The intervals I_c and I_{c-1} both feature positive spare capacity and additionally there is no interval I_{c-2}, or its spare capacity is zero, or I_{c-1} hosts at least one job. In this case, the sum of the spare capacities of I_c and I_{c-1} is considered.

Case 3: $I_p = I_c \cup I_{c-1} \cup I_{c-2}$
The interval I_{c-1} is an empty interval and the intervals I_c and I_{c-2} both feature positive spare capacities. In this case, the spare capacities of all three intervals are considered.

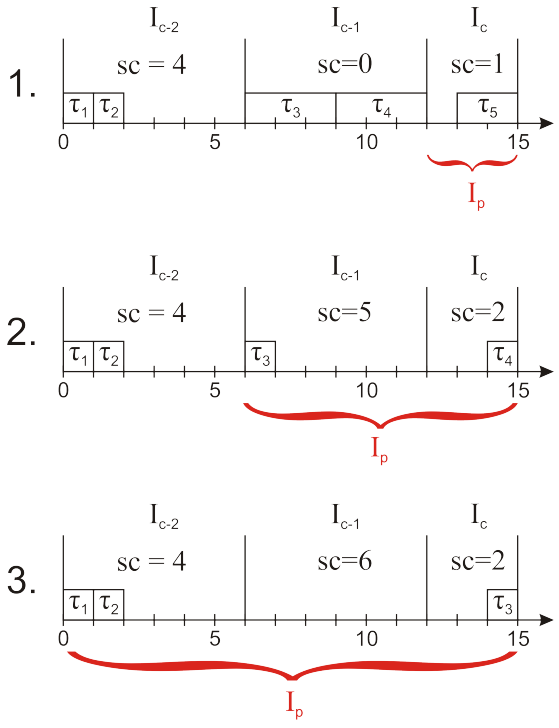


Figure 3.1: *Example showing the different cases for the simple approach.*

The case that $sc(I_c)$ is zero and its predecessor interval features positive spare capacity (or is empty and thus also I_{c-2}'s spare capacity can be considered) has already been considered by the previous iteration of the algorithm and is thus not listed here. Note that under the given assumptions, it is not possible to integrate a non-preemptive aperiodic job such that it covers more than three intervals. In order to span more intervals, all intervals between the first interval and the last interval I_c must be empty to form a sequence of consecutive empty slots for τ_{NP}. By design, the offline phase of slot shifting cannot create two empty intervals next to each other. Thus, the only other way to create a second empty interval would be to start a job prior its assigned interval, which violates the basic assumption from section 3.2.

Remember that under the given assumptions, the spare capacity values of all intervals are non-negative. Thus, the maximum available spare capacity $sc(I_p)$ to accommodate τ_{NP} is given by Equation 3.1 for the different cases:

$$sc(I_p) = \begin{cases} min(sc(I_c), d_{NP} - start(I_c)) & \text{, in case 1} \\ min(sc(I_c), d_{NP} - start(I_c)) + sc(I_{c-1}) & \text{, in case 2} \\ min(sc(I_c), d_{NP} - start(I_c)) + sc(I_{c-1}) + sc(I_{c-2}) & \text{, in case 3} \end{cases} \tag{3.1}$$

If $sc(I_p)$ is larger than or equal to C_{NP}, then the acceptance test for τ_{NP} succeeds. Otherwise, the algorithm shifts the interval I_c to the next interval in time, until the interval with the deadline of the non-preemptive job is reached. If the acceptance test still fails, then τ_{NP} is finally rejected.

There is no need to modify the standard guarantee algorithm to integrate non-preemptive jobs into the schedule. In case of a successful acceptance test, the standard guarantee algorithm is invoked. In case D_{NP} lies in the middle of some interval, this interval will be split into two independent intervals. The offline jobs from the old interval will be moved to the second new interval and the new job will be added to the first new interval. Otherwise, τ_{NP} is added to an already existing interval. In both cases, the guarantee algorithm has to reserve the slots for the non-preemptive job. This is done by updating the spare capacities as described in section 2.3.

Finally, we need to change the runtime mechanisms of slot shifting to ensure the non-preemptive execution of non-preemptive jobs. In the next section, which presents the final approach, we will discuss approaches to do this.

3.2.2 Final Approach

We remove the constraints that applied to the simple approach presented in the previous section. We allow arbitrary scheduling windows for the jobs of offline tasks, i.e., scheduling windows do not have to be identical with to the capacity intervals of the jobs. Also, we do not further restrict the algorithm to offline tables that have been constructed such that all jobs must fit into their corresponding capacity intervals. This implies that intervals may now borrow slots from predecessor intervals to accommodate their jobs.

Since jobs may execute outside their capacity intervals, i.e., possibly much earlier, and since jobs may feature earliest start times (EST) inside their intervals, the acceptance test must be adopted to reflect that.

3.2.2.1 Naive Acceptance Test

A naive approach would simply try to update Equation 3.1, as done below, to allow for intervals with negative spare capacities:

$$sc(I_p) = \begin{cases} max(0, min(sc(I_c), d_{NP} - start(I_c))) & \text{, in case 1} \\ max(0, min(sc(I_c), d_{NP} - start(I_c))) & \\ + max(0, sc(I_{c-1})) & \text{, in case 2} \\ max(0, min(sc(I_c), d_{NP} - start(I_c))) & \\ + max(0, sc(I_{c-1})) + max(0, sc(I_{c-2})) & \text{, in case 3} \end{cases} \tag{3.2}$$

However, this approach will not lead to a suitable solution for integrating non-preemptive tasks. In the following, we will use two counterexamples to exemplify that this naive approach leads to incorrect results: In the first counterexample, one job cannot be shifted freely in its interval, thus the above equation wrongly calculates a higher spare capacity value than is really available for a non-preemptive job. In the second counterexample, one job can start executing much before the start of its interval, thus in reality more spare capacity is available than indicated by the equation.

Counterexample 1 Table 3.1 lists the parameters of a job set consisting of five offline jobs and a single non-preemptive firm aperiodic job. The two resulting intervals and

Job	est	C	d	Type
τ_1	0	1	12	offline
τ_2	0	1	12	offline
τ_3	0	1	12	offline
τ_4	6	3	12	offline
τ_5	12	2	15	offline
τ_{NP}	3	7	15	np aperiodic

Table 3.1: *Parameters of the job set for counterexample 1.*

their jobs can be seen in Figure 3.2. The first interval does not feature enough spare capacity to accommodate the aperiodic job. Thus, the acceptance test presented in the previous section would iterate to the second interval and calculate $sc(I_p)$ according to case 2 with the improved Equation 3.2. As $sc(I_p)$ equals 7, the acceptance test would *succeed* and then the guarantee algorithm would be invoked.

In the given scenario, it is impossible to feasibly integrate τ_{NP} into the given schedule, however. As depicted in Figure 3.2a, starting τ_4 as early as possible does not lead to

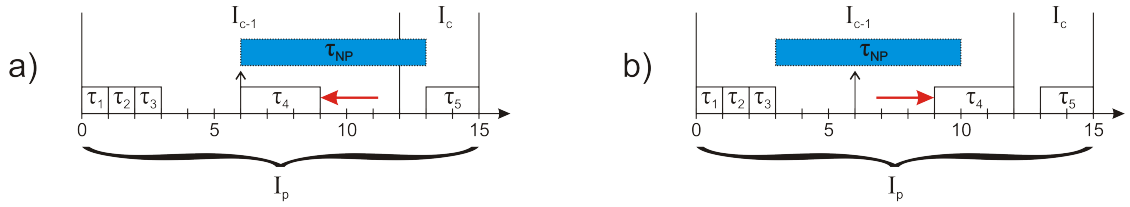


Figure 3.2: *Counterexample 1 for naive acceptance test based on equation 3.2.*

enough empty slots to insert τ_{NP}. The same holds true when τ_4 is started as late as possible, as depicted in Figure 3.2b. In other words, the acceptance test *should fail* in this example, since the non-preemptive job cannot be guaranteed. The problem is that the acceptance test fails to detect that τ_4 cannot be moved freely inside its interval.

Counterexample 2 Table 3.2 lists the parameters of a job set consisting of three offline jobs and a single non-preemptive firm aperiodic job. Figure 3.3a shows the resulting

Job	est	C	d	Type
τ_1	4	2	6	offline
τ_2	6	1	15	offline
τ_3	0	4	15	offline
τ_{NP}	0	8	15	np aperiodic

Table 3.2: *Parameters of the job set for counterexample 2.*

intervals and the jobs. The non-preemptive job τ_{NP} arrives at t = 0 and triggers the acceptance test. The first interval does not offer sufficient spare capacity to accommodate τ_{NP} and the same holds true for the second interval. When the acceptance test considers the third interval, the available spare capacity evaluates to 4 according to Equation 3.2, case 1. Hence, the acceptance test *fails*. However, this is incorrect, since there are more

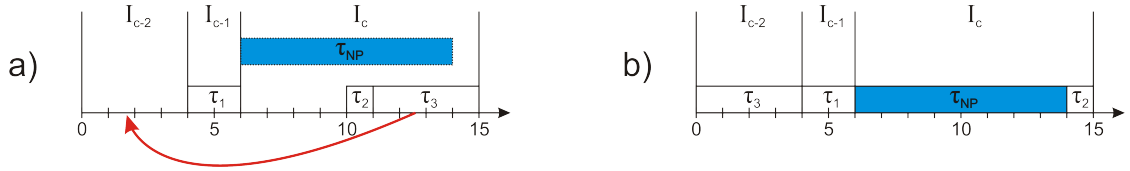


Figure 3.3: *Counterexample 2 for naive acceptance test based on equation 3.2.*

spare capacities available and the acceptance test *should succeed*. If τ_3 starts at t = 0, as depicted in Figure 3.3b, then there exists enough spare capacity to accommodate the non-preemptive job.

3.2.2.2 Final Acceptance Test

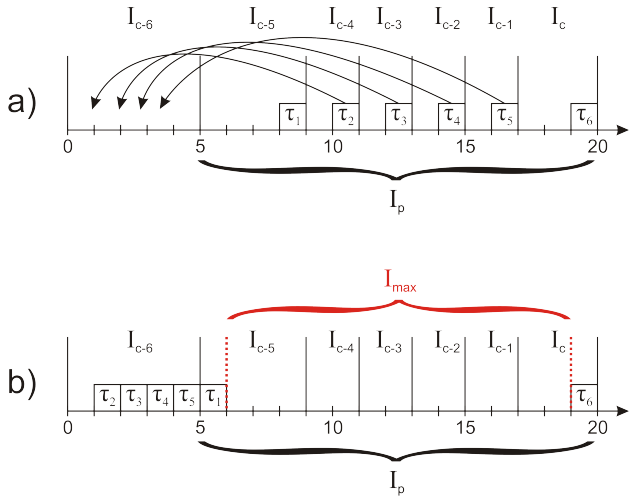
As already explained, the execution of offline jobs and the acceptance test for preemptive aperiodic jobs are performed as described in section 2.3. Only the acceptance test for

jobs of non-preemptive firm aperiodic tasks must be amended. As we have shown before, there exist examples in which a naive spare-capacity-based version fails. Instead, we propose the following algorithm to test a newly arrived non-preemptive firm aperiodic job τ_{NP}. If multiple non-preemptive jobs arrive in a single slot, then we perform the acceptance test in their arrival order. While this might lead to sub-optimal solutions, it significantly reduces the runtime overhead as sorting is avoided.

On non-preemptive aperiodic job arrival, the acceptance test first has to check the trivial condition $d_{NP} \geq t_{now} + C_{NP}$. As before, let the capacity interval under consideration be named I_c. Iterate I_c through all capacity intervals, starting from the current interval in time up to the capacity interval hosting τ_{NP}'s deadline. The set of all previous intervals additionally taken into account to place τ_{NP}, including the interval I_c, is named I_p.

The underlying idea is to calculate the longest possible interval I_{max}, consisting of a continuous sequence of empty slots, within I_p. The algorithm iteratively calculates the delimiters of I_{max}, t_{start} and t_{end}, by shifting all other jobs in the table such that the consecutive number of free slots available for τ_{NP} is maximized. By doing so, the scenarios pointed out by the previously shown two counterexamples will not lead to wrong test results. We will give the exact definition of t_{start} and t_{end} later. If the length of I_{max} is suitable, τ_{NP} is accepted and the guarantee algorithm is invoked. Otherwise, the test iterates to the next interval, i.e., I_c is reassigned. Then, the acceptance test repeats the steps described before until either τ_{NP} is accepted, or the deadline of τ_{NP} is reached and thus, τ_{NP} must be finally rejected.

As in the previous approach, we can identify different cases for I_p, depending on the job set. Lifting the constraints for the final approach does not affect the cases presented before. Additionally, a new case exists: I_p can now span four or more intervals, because jobs may start executing before the start of their interval. To illustrate that, Figure 3.4a shows an example schedule with six jobs. The figure lists the individual job parameters



Task	est	C	d
τ_1	5	1	9
τ_2	0	1	9
τ_3	0	1	12
τ_4	0	1	15
τ_5	0	1	15
τ_6	17	1	20

Figure 3.4: *Example showing I_p spanning six intervals.*

on the right side. When shifting the execution start of the jobs τ_2, τ_3, τ_4, and τ_5, I_p spans six intervals in total, as indicated by Figure 3.4b.

For the first considered interval I_c, i.e., the current interval of time, the start and the end of I_{max} are trivially defined by the current time and the available spare capacity: $t_{start} = t_{now}$ and $t_{end} = t_{now} + sc(I_c)$. The aim is to calculate t_{start} and t_{end} of I_{max} for all other intervals, based on a given scheduling table. Therefore, we need to establish some terms. Figure 3.5 presents an example schedule with a typical scenario. All jobs prior to I_c, except those inside I_c, are assumed to start as early as possible, i.e., at their earliest start time. If multiple jobs feature the same earliest start time, as shown with jobs τ_3, τ_4, and τ_5 in Figure 3.5, then a chain of jobs will be formed.

We first define a set B of jobs in Equation 3.3. This set contains all jobs that feature earliest start times and deadlines before the start of the currently considered interval I_c. Furthermore, their executions start at their earliest start times.

$$B = \{\tau_i | (est_i = start(\tau_i)) \wedge (d_i \leq start(I_c)), \forall i\} \tag{3.3}$$

The actual start of the execution of a job is calculated recursively by Equation 3.4: The execution of a job starts either at its earliest start time or after the previous job finished, whichever is later.

$$start(\tau_i) = max\{est_i, (start(\tau_{i-1}) + C_{i-1})\} \tag{3.4}$$

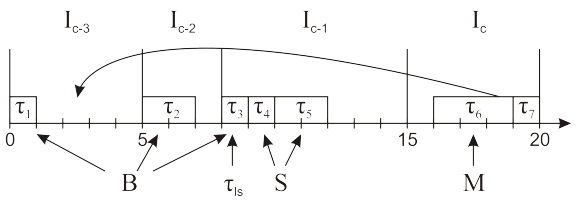
In Equation 3.5, we identify the last job to start among the jobs from the set B: τ_{ls}.

$$\tau_{ls} \in B : est_{ls} > est_i, \forall \tau_i \in B \setminus \{\tau_{ls}\} \tag{3.5}$$

So far, only the jobs prior to I_c have been considered. However, there are scenarios in which also jobs from inside I_c have to be shifted. We define a set M of jobs which belong to I_c and whose shifting to their earliest start times increases the length of the interval I_{max}:

$$M = \{\tau_i | (dl(\tau_i) = end(I_c)) \wedge (est(\tau_i) < est(\tau_{ls}))\} \tag{3.6}$$

As long as M is non-empty, the following approach is used: The job of M with the smallest sum of its earliest start time and its worst case execution time is selected to



Task	est	C	d
τ_1	0	1	5
τ_2	5	2	8
τ_3	8	1	15
τ_4	8	1	15
τ_5	8	2	15
τ_6	1	3	20
τ_7	15	1	20

Figure 3.5: *Example showing the sets B, S, M, and the job τ_{ls}.*

start as early as possible. The sets B and M as well as τ_{ls} and the spare capacities in the affected intervals are updated and these steps are repeated until M is empty. To avoid the sorting and decrease the runtime overhead, any job in M could be selected. This is because M is defined such that the length of the interval I_{max} increases, independent of which job of M is shifted to its earliest start time. Nevertheless, by always first selecting the job with the smallest sum of its earliest start time and its worst case execution time, the length of I_{max} will increase profoundly. By shifting a job from M, it is possible that its original interval becomes an empty interval. To save runtime overhead, this empty interval is not joined with its successor interval.

As shown in Figure 3.5, the job τ_{ls} can be succeeded by several other jobs which do not start exactly at their start time. To identify these jobs, we define the set S of the "succeeding jobs" united with job τ_{ls}:

$$S = \left\{\tau_i \middle| (est_{ls} \leq est_i) \wedge (d_i \leq start(I_c)), \forall i\right\} \tag{3.7}$$

t_{start} calculates as the earliest start time of job τ_{ls} plus the worst case execution times of all jobs in S. As slot shifting is a work conserving scheduling algorithm, there cannot be any idle slots after the start of τ_{ls} until the last job of S finishes its execution. Thus, t_{start} is given by:

$$t_{start} = est_{ls} + \sum_{\tau_i \in S} C_i \tag{3.8}$$

and t_{end} is calculated as:

$$t_{end} = start(I_c) + min\left\{sc(I_c), \left(d_{NP} - start(I_c)\right)\right\} \tag{3.9}$$

Once these delimiters are known, the final condition for the acceptance test can be stated as:

$$\begin{aligned} &C_{np} \leq t_{end} - t_{start} \\ &C_{np} \leq start(I_c) + min\left\{sc(I_c), d_{np} - start(I_c)\right\} - est_{ls} - \sum_{\forall \tau_i \in S} C_i \end{aligned} \tag{3.10}$$

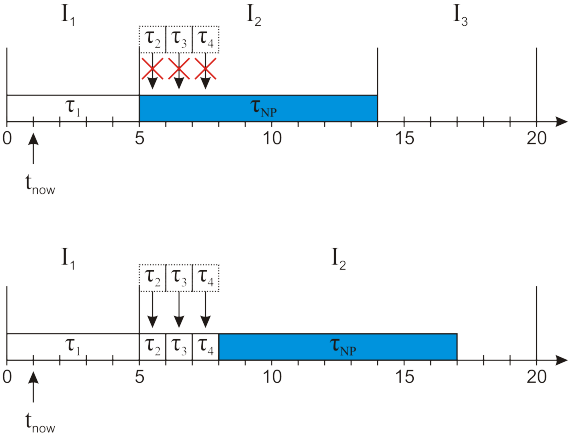
3.2.2.3 Guarantee Algorithm

The essential difference between handling preemptive and non-preemptive aperiodic jobs lies in the method to *identify* a suitable position in the schedule for the job. Once the acceptance test succeeds, updating the scheduling table as described in section 2.3.2 works for preemptive as well as for non-preemptive aperiodic jobs: If needed the guarantee algorithm splits an interval I_x into two intervals I_{x1} and I_{x2}, adds the aperiodic job to I_{x1}, and shifts the jobs of the original interval to I_{x2}. Otherwise, the aperiodic job is added to an already existing interval I_x. In both cases, the spare capacities of the affected intervals are updated afterwards. This ensures sufficient slots for the execution of the aperiodic job and anticipates that succeeding aperiodic jobs claim these slots.

This "standard" guarantee algorithm effectively integrates all aperiodic jobs as late as possible to leave room for future aperiodic jobs. If an aperiodic job τ_x is integrated

and then more aperiodic jobs arrive, they thus can be integrated using this flexibility. Nevertheless, slot shifting is EDF-based, i.e., a work conserving algorithm. So, if after τ_x no other aperiodic jobs arrive and τ_x has the earliest deadline of all ready jobs, then the scheduler will select the aperiodic job for execution as soon as possible, i.e., in this case right away. This flexible approach improves the responsiveness of aperiodic jobs at runtime.

The guarantee algorithm for non-preemptive aperiodic jobs can be tuned, i.e., the responsiveness of non-preemptive jobs can be improved at the cost of flexibility. Instead of as late as possible, the guarantee algorithm can integrate the non-preemptive jobs as soon as possible. To achieve this, the algorithm could artificially shorten their deadlines. The new deadline calculates as the current time plus the worst case execution time of the non-preemptive job. Furthermore, the algorithm needs to add some additional time[1] if other jobs in the schedule enforce a delayed start of the non-preemptive job: $d_{NP}^{new} = t_{now} + C_{NP} + t_{add}$. While this method would lead to quick reactions to non-preemptive jobs, it would at the same time come at the price of a potentially degraded overall acceptance ratio of the system. Figure 3.6 illustrates this trade-off between providing flexibility for future aperiodic jobs and improving the response time for the non-preemptive jobs. On the right side of the figure, the job set properties are listed. The figure shows the resulting schedule using the standard (bottom) and the modified guarantee algorithm (top): The former algorithm does not alter the original deadline of the non-preemptive job and thus at runtime jobs τ_2, τ_3, and τ_4 can be integrated into the schedule. The modified guarantee algorithm shortens the deadline of τ_{NP} to 14 and creates a new interval to host τ_{NP}. Since τ_{NP} cannot be moved, the three later arriving aperiodic jobs are rejected.



Task	est	C	d
τ_1	0	5	5
τ_2	2	1	6
τ_3	2	1	7
τ_4	2	1	8
τ_{NP}	1	9	20/14

Figure 3.6: *Example illustrating the trade-off between providing flexibility for future aperiodic jobs and improving response time for the non-preemptive jobs.*

Both variants of the guarantee algorithm ensure the correct integration of non-preemptive jobs. The latter method is less flexible but guarantees the shortest possible response time of non-preemptive jobs. The price is a significantly reduced flexibility,

[1] This additional time can be easily calculated during the acceptance test.

however. This approach should be preferred if good responsiveness of non-preemptive jobs has higher priority than maximizing the acceptance ratio.

The first (standard) method should be preferred if more flexibility to add other aperiodic jobs is more important than optimum responsiveness of the non-preemptive job. Although this approach is more flexible, none of the presented approaches are optimal. In fact, for both approaches counterexamples can be constructed such that while a non-preemptive job is executing, other aperiodic jobs are rejected.

3.2.2.4 Non-Preemptive Execution

The runtime mechanisms of slot shifting must be modified to ensure that, whenever the scheduler selects a non-preemptive job τ_{NP} to execute next, this job really executes non-preemptively. The simplest approach would be to set the available spare capacity temporarily to zero during execution of τ_{NP}. However, this only emulates a non-preemptive execution of τ_{NP}. At every slot boundary, the scheduler would still wake up, perform the acceptance test (and potentially the guarantee algorithm) for newly arrived aperiodic jobs. Finally, the scheduler would select τ_{NP} for execution, as the spare capacity is temporarily set to zero. Accordingly, this approach suffers from non-negligible interference by the scheduler at every slot boundary.

A cooperative approach would disable all interrupts and then schedule the non-preemptive job. Obviously, this approach suffers from the risk that τ_{NP} overruns. Even worse, τ_{NP} could never return control to the scheduler by entering a never-ending loop, thus leading to system failure.

The best approach is to program a hardware timer to fire at the end of the expected execution of τ_{NP}. Then, all interrupts except for the one triggered by the timer are masked out. By this means, τ_{NP} is executed non-preemptively and it is ensured that the scheduler eventually gains control again.

3.3 Example

This example consists of four offline jobs τ_W, τ_X, τ_Y, and τ_Z and one non-preemptive aperiodic job τ_{NP}. Table 3.3 lists the earliest start times, worst case execution times, and the deadlines of the jobs.

Job	est	C	d	Type	Interval
τ_W	0	1	5	offline	I_1
τ_X	0	3	7	offline	I_2
τ_Y	0	2	11	offline	I_3
τ_Z	13	1	16	offline	I_5
τ_{NP}	1	9	16	aperiodic	–

Table 3.3: *Job properties of example schedule.*

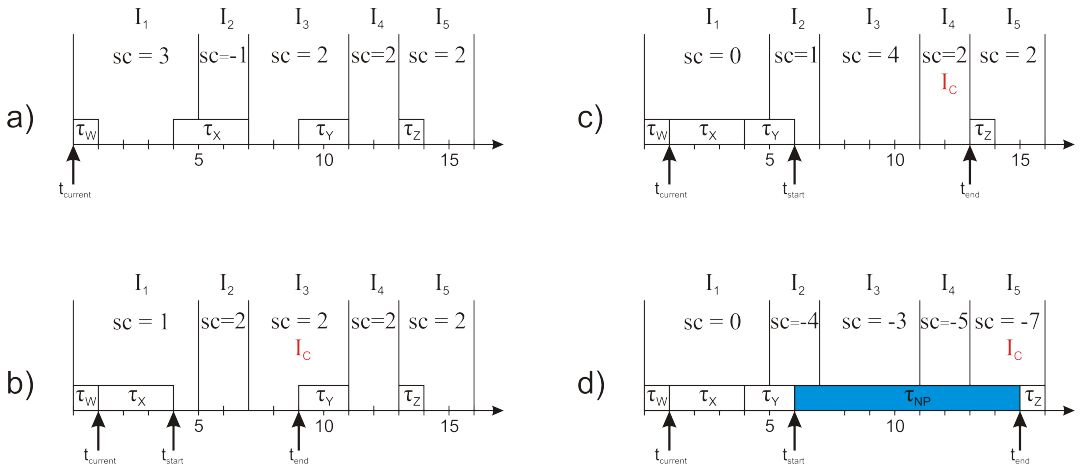


Figure 3.7: *Example schedule with five jobs.*

Figure 3.7a presents the original offline schedule, which is the basis for our algorithm as described in section 3.2.2: The first interval I_1 has a length of five slots and contains τ_W with a worst case execution time of 1. Interval I_1 lends one slot to its successor interval, thus the spare capacity of interval I_1 is calculated to be 3. The next interval contains τ_X and makes use of the borrowing mechanism, hence the spare capacity of I_2 is -1. The third interval holds τ_Y with a worst case execution time of 2, which leads to a spare capacity of 2. Interval I_4 is empty and the last interval, I_5, accommodates τ_Z with worst case execution time of 1 and hence has a spare capacity of 2.

When $t_{current}$ equals 1, an aperiodic non-preemptive job τ_{NP} arrives with deadline 16 and worst case execution time of 9. In the first step, the algorithm defines $I_c = I_1$, $B = \{\tau_W\}$, $\tau_{ls} = \tau_W$, $M = \{\}$, $S = \tau_W$, $t_{start} = 1$, and $t_{end} = 4$. Since the time-span between t_{start} and t_{end} is not enough for job τ_{NP}, the algorithm defines $I_c = I_2$ in the next step. The algorithm recalculates B, τ_{ls}, M, and S, which all do not change. The job τ_{NP} cannot be scheduled starting immediately at t = 4 without τ_Y missing its deadline, thus I_c is reassigned to interval I_3. Now, jobs τ_W and τ_X start as early as possible, the spare capacities are updated and $B = \{\tau_W\}$, $\tau_{ls} = \tau_W$, $S = \{\tau_W, \tau_X\}$, $M = \{\}$, $t_{start} = 4$, and $t_{end} = 9$, see Figure 3.7b.

Due to the lack of sufficient successive slots to accommodate τ_{NP}, the algorithm repeats these steps and hence, prepones the start of the execution of τ_Y in the same manner as τ_X when it shifts I_c to the next interval I_4. The updated spare capacities and the values for t_{start} and t_{end} are depicted inFigure 3.7c: There are not enough consecutive empty slots to accommodate τ_{NP}.

In the final step, the algorithm assigns I_c to the last interval and sets t_{end} to 15, adds τ_{NP} to the schedule, and updates all values as shown in Figure 3.7d.

3.4 Discussion

In this chapter, we presented our approach to feasibly integrate jobs of non-preemptive firm aperiodic tasks into the schedule at runtime. As the original slot shifting algorithm described in chapter 2, our approach is divided into an offline and online phase.

Our approach uses the same unmodified offline phase as described in section 2.2. We will now summarize the properties of the online phase of our approach.

The first major change affects the acceptance test. In general, our approach to handle non-preemptive firm aperiodic jobs features a runtime complexity of $O(n^2)$, with n being the number of jobs in the job set. As the original acceptance test, which is of linear complexity, our acceptance test iterates through the intervals up to the deadline of the aperiodic job. While doing this, each job can be shifted to make room for the new aperiodic job. In the worst case however, shifting a job triggers the recalculation of the spare capacities of the intervals and the recalculation of the sets B, M, S, and the job τ_{ls}. Thus, jobs are considered again for shifting, even if they are not actually shifted again.

One of the benefits of our approach is that no new offline scheduling table needs to be calculated. Instead, the information found in the existing table is used to perform the acceptance test. Then, the table is updated to reflect the presence of the new job and to reserve the empty slots for the execution of the job. This implies another benefit of our approach: The presented acceptance test only has a very low memory overhead. In essence, it is sufficient to store a simple pointer to represent each job in one of the sets.

Nevertheless, the capability to handle non-preemptive jobs comes at a price. The algorithm to identify a consecutive set of empty slots suitable for the non-preemptive job increases the complexity of the acceptance test compared to the original acceptance test. Furthermore, adding support for non-preemptive jobs unavoidably increases the risk of a decreased overall acceptance ratio for aperiodic jobs.

As we have seen before, there is no need to modify the guarantee algorithm. Nevertheless, by doing so, we can fine-tune the behavior of the slot shifting algorithm. This enables the system designer to trade flexibility for future aperiodic jobs against improved responsiveness of the non-preemptive jobs. We believe that this trade-off needs to be decided depending on the specific requirements of the project and given constraints.

The final modification affects the implementation of the system, more precisely of the runtime scheduler: In order to yield full support for non-preemptive jobs, the runtime scheduler must not allow for any interferences, e.g., by interrupt service routines.

Chapter 4

Event-Triggered Activities in Time-Triggered Schedules on Multicore Systems

In this chapter, we introduce and describe our approach to global slot shifting algorithms for multicore systems. We start the chapter with a brief introduction. The introduction includes a description of the challenges that are associated with global slot shifting algorithms. We discuss the issues and bottlenecks of different methods to tackle the problem. After that, we present two global algorithms: a spare-capacity-based and a negotiation-based version of the slot shifting algorithm. In the succeeding chapters 5 and 6, we will analyze the different slot shifting versions in great detail in different experiments.

4.1 Introduction

As detailed in chapter 2, the slot shifting algorithm is designed as a preemptive algorithm for distributed systems. The algorithm aims at feasibly integrating aperiodic tasks into the local schedule at runtime. In this chapter, we present our approach to extend the original slot shifting algorithm to multicore architectures. First, we will discuss the challenges that arise due to the change from a distributed system to a multicore architecture. The basic idea of our new global algorithms is to make use of the given computational resources in a multicore system to accommodate aperiodic tasks and to improve the overall system utilization. After that, in the section named "Methodology" we present two different global slot shifting algorithms. The first bases its decisions on the available spare capacities on the other cores in the system. The second algorithm uses a negotiation-based approach to identify suitable cores in the system.

4.1.1 Challenges

In the chapter "Background and Related Work", we discussed the results for real-time scheduling on multiprocessor systems. As already pointed out, the results obtained from multiprocessor scheduling can be directly applied to multicore systems: multicore architectures are just a specific implementation of multiprocessor systems. The fundamental result is that there exists no optimal online multiprocessor real-time scheduling algorithm for aperiodic tasks, without being clearvoyant to future aperiodic task arrival times [45, 46]. With that knowledge, our focus is set to an algorithm for multicore systems that yields "good" results and features acceptable runtime overheads.

For performance reasons, any multicore slot shifting algorithm must aim at first trying to integrate aperiodic tasks locally. When the situation occurs that the local acceptance test fails, the challenge arises to identify the best matching core for the aperiodic task. Thus, each core must have access to information about the individual state of the other cores in the system. This challenge can be subdivided into multiple issues: first, what metric to base the decision making process on, i.e., which information is meaningful to represent the state of the other cores. Second, how to efficiently exchange and store data needed to perform the decision making process. Third, how to implement it such that the runtime overhead is minimal.

To establish any metric, data exchange and thus synchronization among the cores is required. Special care needs to be taken to avoid unpleasant synchronization penalties that thwart the benefits of the multicore system. Thus, the data exchange must be reduced to an acceptable minimum that on the one hand enables establishing a meaningful metric to base migration decision on, and that on the other hand reduces the synchronization penalty. Highly connected to this challenge is the question on where and how to store the information that allows to obtain a state of the individual cores. A globally shared central memory location intuitively increases synchronization costs and risks, while distributed storage increases the memory demand and local management overheads.

4.2 Methodology

We consider scenarios, in which at any time one or several jobs of aperiodic tasks may arrive at random cores. For simplicity reasons, we will in the following sections describe our algorithms for a maximum of one aperiodic job arrival per core per slot. In other words, multiple jobs of aperiodic tasks can arrive per slot, but not on the same core. Nevertheless, these algorithms are not limited to such scenarios.

The simplest approach to tackle this scenario is a partitioned implementation of the slot shifting algorithm. The assignment of the tasks to the cores is performed offline and during the online phase, each core runs the slot shifting algorithm independently. Aperiodic jobs arrive at the individual cores at runtime, are tested and possibly integrated. While this approach poses only minimal implementation challenges, it suffers from the drawback of limited resource utilization and minimal flexibility.

Another approach is global scheduling. If an aperiodic job fails the local acceptance test, then it is desirable to delegate this job to another core with sufficient resources, i.e., spare capacities. From the point of view of the delegating core, the main problem is: How to decide to which core to delegate an aperiodic job in order to increase the likelihood of successful acceptance. There exists no optimal multicore online scheduling algorithm for aperiodic tasks, as has been proved in [45].

Even after a promising core has been found, it is not ensured that migrating the job to that core will result in successful integration. An acceptance test on the receiving core has to ensure that the aperiodic job meets its deadline without violating already guaranteed jobs scheduled on that core. Otherwise, the job cannot be guaranteed on that core and has to be further delegated to another core. In the worst case, an incoming aperiodic job cannot be accepted, e.g., due to a simultaneously locally arrived and accepted aperiodic job. Thus, the delegated job needs to be delegated further, which increases the risk of missing its deadline.

To find a matching core, the delegating core needs to know the amount of leeway in the local scheduling tables of all other cores. As described in chapter 2, the scheduling tables and also the amount and location of spare capacities have been determined in the offline phase for all cores and are maintained online. So, one approach to find a matching core is to store this information about all cores of the system locally on each core and keep it up to date at runtime. This approach suffers from increased memory demands and computational overhead on each core and requires synchronized exchange of update messages between cores, when new aperiodic jobs are integrated—a bottleneck which limits scalability.

Another approach to find a matching core is to globally store the interval and available spare capacity information for all cores in a memory shared by all cores and manage it there. The efficiency of this approach is limited as access to the shared data needs to be restricted to maintain data integrity—one core could slow down all other cores and thus degrade the overall system performance.

An orthogonal problem is to minimize the runtime overhead caused by determining to which core to delegate an aperiodic job and the overhead caused by checking for and receiving the new incoming delegated aperiodic jobs.

For these reasons, we use a different approach: As mentioned in section 1.3, there exists no optimal online multiprocessor scheduling algorithm for aperiodic jobs, thus we developed two global heuristic algorithms based on the original slot shifting algorithm. Both algorithms have in common that if the local acceptance test for an aperiodic job fails on a core, they delegate the job to another core[1] which will perform an acceptance test in the next slot.

On the one hand, by not directly interrupting the other core, preemptions at arbitrary moments in time are avoided. Thus, there are no preemptions in the middle of the execution of some job on the other core, i.e., job executions for a fraction of a slot do not occur, and there are no preemptions while executing the slot shifting algorithm itself. From the global point of view, the impact of this single slot delay of an aperiodic

[1] Flags ensure that aperiodic jobs are not delegated multiple times to the same core and that they are guaranteed only on a single core.

job on the overall acceptance ratio is limited[2].
On the other hand, this slightly delayed acceptance test also has a drawback: In the worst case aperiodic jobs that could be guaranteed on some other core are rejected because they might get delegated multiple times before arriving at a suitable core and finally miss the deadline. Thus, even after the problem of assigning a suitable core to the aperiodic job has been solved, the acceptance test might fail.

4.2.1 Global Algorithm Based on Spare Capacity

If local acceptance fails, this global algorithm uses the available spare capacities on other cores as metric to base the decision on, to which core to delegate an aperiodic job. An overview of the algorithm can be found in Listing 4.1. After constructing the initial ready list, in the first step this global algorithm checks whether a new aperiodic job has arrived. In that case, it performs a local acceptance test. If there is not sufficient spare capacity available to meet the aperiodic job's execution requirements, then the algorithm delegates this job to the core with the highest available spare capacity until the deadline of the job. In other words, this necessary delegation of aperiodic jobs to other cores is based on a first-fit heuristic. As already explained in section 2, every core keeps track of its available spare capacities. Thus, it adds little overhead to make this information accessible to all other cores in a shared memory.
To delegate the aperiodic job, the delegating core updates a table in the shared memory with the job's parameters and finally sets a notify flag for the other core. At the beginning of every slot, every core checks whether its notify flag is set. If there are delegated jobs pending, then the core first acquires the delegated jobs' properties from the table. After that, the corresponding core performs acceptance tests for the aperiodic jobs. Consequently, every delegation of an aperiodic job delays its possible acceptance by one slot, as already discussed.
In the case of a local aperiodic job arriving in parallel with a delegated job, the situation might arise that accepting one of them locally makes the other one or a possible future aperiodic job missing its deadline. Since it has been proven that there exists no optimal solution without being clairvoyant [45, 46], we break this tie in favor of the delegated aperiodic job, as already resources have been used on other cores. In case one or multiple acceptance tests fail, delegation to another core takes place. A flag in the properties of the job keeps track of the already visited cores to avoid loops or unnecessary multiple acceptance tests on the same core.

If the local acceptance test succeeds, the guarantee algorithm integrates the aperiodic job and updates the spare capacities to reflect that. After that, the available spare capacity value in the shared memory is updated. By storing the available spare capacities for each core in the shared memory, this algorithm tries to minimize repeated delegations. This increases the chances to accept aperiodic jobs but does not guarantee their acceptance. The time to calculate the spare capacity values is negligible since in the original slot shifting algorithm every core updates its spare capacities, i.e., they are

[2] We assume most aperiodic jobs' slack to be much larger than zero slots as they cannot be guaranteed on other cores otherwise.

Listing 4.1: *Global spare-capacity-based Slot Shifting (Global Algorithm 1).*

```
1  initialization: construct ready list;
2  for each slot {
3      if(new aperiodic jobs arrived) {
4          create sorted list of aperiodic jobs: delegated jobs first;
5      }
6      for all new aperiodic jobs {
7          acceptance test;
8          if(successful) {
9              add job to local schedule;
10             update spare capacities(local);
11             update spare capacities(shared mem);
12         }
13         else {
14             check job delegation history;
15             find possible target cores;
16             get spare capacities of target cores;
17             find core with max spare capacity;
18             delegate job;
19         }
20     }
21     update ready list;
22     select next job to run;
23     if (selected job executes outside its interval) {
24         update spare capacities(local);
25         update spare capacities(shared mem);
26     }
27     schedule next job;
28 }
```

already available. Storing this information in the shared memory adds some overhead (see results in section 5.3).

This algorithm is a compromise between quickly selecting a core and spending more computational resources to find the best matching core as proposed in the next algorithm.

4.2.2 Example Schedule for Global Algorithm 1

In this section, we present an example to demonstrate the runtime behavior of global algorithm 1. The example has originally been designed to test the correctness of our implementation of the algorithm in a four core scenario. It consists of four offline and four aperiodic jobs whose parameters are listed in Table 4.1. All aperiodic jobs are mapped to cores that do not feature sufficient spare capacity to integrate them locally.

Hence, the acceptance tests do fail and every job is delegated for further testing to the core with most available spare capacity. The deadlines of jobs τ_5, τ_6, and τ_8 are chosen such that only a single delegation is possible without the job missing its deadline. The deadline of τ_7 has been chosen longer than the deadline of τ_4 to test whether the scheduler correctly implements the EDF algorithm[3].

For example, job τ_5 arrives at t = 1 at core 1. It requires five slots to execute but the schedule on this core only offers a spare capacity of 2. Thus, the acceptance test fails and the available spare capacities of the other cores are acquired and compared:

2. core: 5 slots
3. core: 4 slots
4. core: 3 slots

Then, τ_5 is delegated for the acceptance test to core 2, which features most spare capacity. At the beginning of the slot (at t = 2), core 2 receives the parameters of τ_5. Additionally, in this slot job τ_6 arrives locally at core 2. Slot shifting favors the delegated job τ_5 over τ_6 and successfully performs the acceptance test for it. Core 2 integrates it into the schedule and then triggers the acceptance test for τ_6, which fails. After comparing the available spare capacities of the other cores (0, 4, and 3 slots, respectively), slot shifting delgates τ_6 to core 3. Finally, slot shifting selects τ_5 to execute next as it features an earlier deadline than τ_2. Similarly, slot shifting locally tries to integrate τ_6, τ_7, and τ_8 before they are sent to another core and accepted there. Figure 4.1 shows the resulting correct schedule.

Job	Core	est	C	d	Type
τ_1	1	0	8	10	offline
τ_2	2	0	5	10	offline
τ_3	3	0	6	10	offline
τ_4	4	0	7	7	offline
τ_5	1	1	5	7	aperiodic
τ_6	2	2	4	7	aperiodic
τ_7	3	3	3	10	aperiodic
τ_8	4	4	2	7	aperiodic

Table 4.1: *Job properties of example schedule.*

4.2.3 Global Algorithm with Negotiation-Based Acceptance Test

The decision making process of the second global algorithm is based on negotiation among the cores. Thus, this algorithm does not require the information about each

[3] In an incorrect implementation, τ_7 would directly start executing (at t = 4) after its delegation to core 4 for three slots resulting in τ_4 missing its deadline.

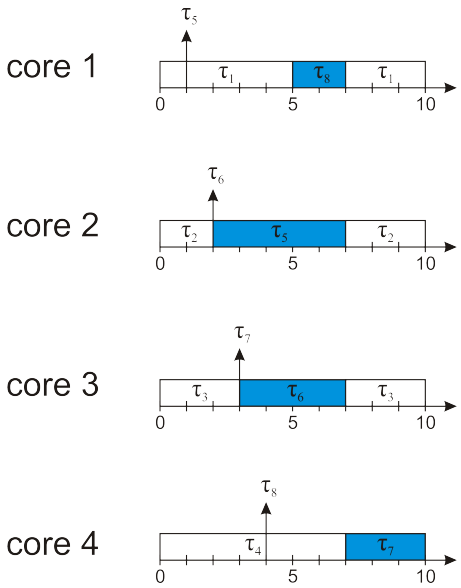


Figure 4.1: *Example schedule for global algorithm 1.*

core's available spare capacities. It thus saves the overhead corresponding to updating/ retrieving the information from the shared memory. An overview of algorithm 2 can be found in Listing 4.2. This algorithm makes use of all other available cores in the system if a local acceptance test fails: instead of making the core on which the test failed search for a good core to delegate the aperiodic job to, this algorithm triggers all other cores to perform an acceptance test for its locally rejected aperiodic job. In case multiple cores can accept the aperiodic job, the fastest core wins. The difference to the previous algorithm is that not a specific core starts the acceptance test, but all remaining cores in the system. This minimizes the response time of the aperiodic job compared to the previous global algorithm, since multiple delegations of the same job are efficiently avoided.

The algorithm maintains two tables in the shared memory: the first table, the IN-table, to store the jobs which have been rejected in the previous slot by each core. In the current slot, the cores perform acceptance tests for all the aperiodic jobs in this table. The first core that successfully tests an aperiodic job integrates it into the local schedule. All other cores continue with testing the next aperiodic jobs, if there are multiple in the IN-table. At the end of the acceptance tests, the table is erased.
In the second table, the OUT-table, all cores store the aperiodic jobs which they reject in the current slot. The algorithm sets a notify flag to trigger acceptance tests for these jobs in the next slot. When the next slot starts, the current OUT-table becomes the IN-table.
A semaphore ensures that a job cannot be guaranteed by more than one core, even if multiple acceptance tests succeed.

For now, we restrict ourselves to scenarios where a single aperiodic job can arrive per slot and core locally, i.e., up to N aperiodic jobs per slot. In the worst case, up to N acceptance tests for each aperiodic job take place, where N is the number of cores.

Listing 4.2: *Global negotiation-based Slot Shifting (Global Algorithm 2).*

```
1  initialization: construct ready list;
2
3  for each slot {
4      if(new aperiodic jobs arrived) {
5          create sorted list of aperiodic jobs: delegated jobs first;
6      }
7      for all aperiodic jobs {
8          acceptance test;
9          if (successful) {
10             if (local core is first core to accept) {
11                 guarantee job locally;
12                 update spare capacity(local);
13             }
14         }
15         else {
16             reject job;
17         }
18     }
19     update ready list;
20     select next job to run;
21     if (selected job outside its interval) {
22         update spare capacity(local);
23     }
24     schedule next job;
25 }
```

Since each of the N cores could have rejected an aperiodic job in the previous slot, and additionally, on each of the cores, a new aperiodic job might arrive, a total of 2N jobs can arrive per slot. So in the worst case, per slot up to $2N^2$ acceptance tests take place in the whole system. Nevertheless, in a real implementation that cannot be perfectly synchronized, the number of acceptance test per job will be less than N in the average case: once a job is accepted by one core, this job will be removed from the list for testing. On systems with tens or hundreds of cores, limiting the number of participating cores to smaller clusters can solve the issue of limited scalability.

One benefit of this algorithm is that multiple delegations of the same aperiodic job are completely avoided. This saves the overhead to determine multiple times the best core to delegate the aperiodic job to. Another benefit is that if there exists a core that can accept the job, then this algorithm will directly find it, i.e., the response time is minimized. In the following section, we present an example schedule, in which this negotiation-based slot shifting outperforms the spare-capacity-based slot shifting.

4.2.4 Example Schedule for Global Algorithm 2

In this section, we present an example to contrast spare-capacity-based slot shifting with negotiation-based slot shifting. The offline table features six jobs, $\tau_1 - \tau_6$, assigned on four cores. Table 4.2 lists the properties of the jobs. Additionally, at t = 1 a single aperiodic job, τ_7, arrives on core 1. Note that none of the offline jobs in this example offer

Job	Core	est	C	d	Type
τ_1	1	0	8	15	offline
τ_2	2	0	2	2	offline
τ_3	2	4	2	6	offline
τ_4	3	0	3	3	offline
τ_5	3	4	2	6	offline
τ_6	4	6	9	15	offline
τ_7	1	1	3	6	aperiodic

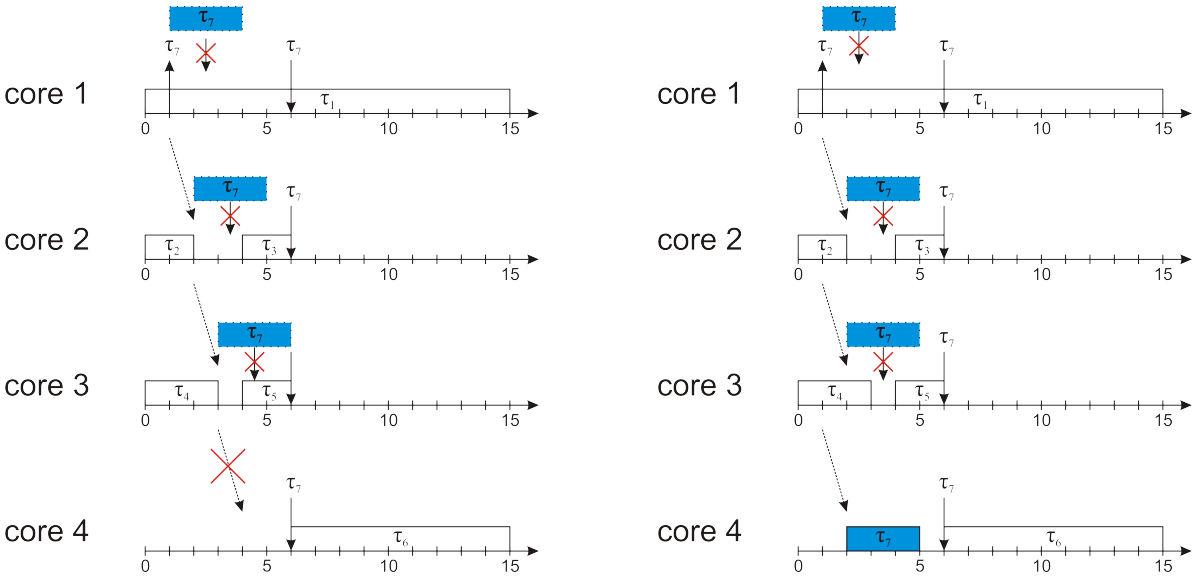
Table 4.2: *Job properties of example schedule.*

any flexibility, i.e., they cannot be shifted. Based on the properties given in Table 4.2, the individual scheduling tables on the cores feature the following spare capacities: 0, 11, 10, and 6 slots, respectively.

Figure 4.2a shows the resulting schedule when spare-capacity-based slot shifting is employed. As core 1 does not offer any spare capacity, the local acceptance test fails. Consequently, the aperiodic job τ_7 is delegated to core 2 which features most available spare capacity. At t = 2, the acceptance test for τ_7 on core 2 fails, as there is not sufficient spare capacity available up to the deadline of τ_7. In a final attempt to accommodate the job, τ_7 is sent to core 3. At t = 3, core 3 performs a local acceptance test for τ_7, which fails for the same reason: insufficient spare capacity up to the deadline of τ_7. This reject is final for τ_7, as τ_7 would definitely miss its deadline by the time of its arrival on another core. In other words, at t = 4 the trivial condition $t_{now} + d_7 \geq C_7$ would be violated, so any further delegation attempt is futile.

This example shows, that a greedy heuristic not always succeeds in time to identify the correct core to send the aperiodic job to. The "flaw" occurs because after the local test on core 1 failed, on each other core all the available spare capacity in the future has been taken into account. If only the available spare capacity on each core from the earliest start time of τ_7 up to its deadline had been considered (0, 2, 1, and 5 slots, respectively), then τ_7 would have been directly sent to core 4 and successfully integrated. Our implementation of the spare-capacity-based slot shifting algorithm that is used in the succeeding chapters to perform the experiments has been modified to avoid this problem.

Figure 4.2b shows the resulting schedule when negotiation-based slot shifting is applied. After the local acceptance test on core 1 failed, it triggers all other cores to perform a local acceptance test for τ_7 in the next slot. At t = 2, these tests are concur-



(a) *Global algorithm 1 fails to accept aperiodic job τ_7.*

(b) *Global algorithm 2 integrates aperiodic job τ_7.*

Figure 4.2: *Different resulting schedules for the same job set.*

rently performed on core 2, 3, and 4. Only core 4 offers sufficient spare capacity and thus successfully integrates τ_7 into the local schedule.

Chapter 5

Efficiency Evaluation

"*Efficiency is doing things right; effectiveness is doing the right things.*"
Peter Ferdinand Drucker

This chapter describes the experiments that we conducted to perform an extensive evaluation of the slot shifting algorithms on multicore systems. This chapter details the evaluation of the efficiency of the slot shifting algorithms. The evaluation of the effectiveness of the slot shifting algorithms can be found in the next chapter. Both chapters include a description of the implementation, of the performed experiments, and a discussion of our findings.

In this chapter, our aim is to thoroughly analyze the efficiency of slot shifting, i.e., the overheads and the runtime costs associated with the partitioned and the two global slot shifting algorithms. Additionally, we modify the existing guarantee algorithm and evaluate the properties of the resulting algorithm.

We analyze how much computational effort is spent for the goal of integrating randomly arriving jobs of aperiodic tasks feasibly into the offline derived scheduling table. As already explained, this scheduling table is used to schedule the jobs of the offline guaranteed tasks at runtime of the system. Our analysis shows and relates the costs in terms of runtime for different operations of the different slot shifting algorithms. The ultimate goal is to provide a holistic view of the runtime costs to allow the reader to evaluate the efficiency of the different slot shifting algorithms.

To obtain reproducible and thus reliable measurements of the runtime overheads, we employ a cycle-accurate multicore simulator: MPARM, the features and properties of which are presented in section 5.1. We perform experiments with selected mixed job sets consisting of jobs of aperiodic and offline guaranteed tasks and measure the runtime. These task sets trigger different behavior of the algorithms. We categorize and evaluate the measured runtimes for different experiments. Also, based on these measurements we analyze the runtime overheads of the different sub-functions of our slot shifting implementation. This allows to determine the overheads and the costs associated with different operations, such as performing one or multiple acceptance tests, or splitting of intervals. We analyze the different experiments that trigger different behavior of the slot shifting code and highlight where different overheads originate from. Furthermore, we modify the acceptance test to feasibly shorten the deadlines of aperiodic jobs. As our experiments show, this improves the runtime of the algorithm and to

increase the responsiveness of the aperiodic jobs. Additionally, we evaluate the impact of this modification on the measured runtime of the algorithm.

The remainder of this chapter is structured as follows: First, we describe the MPARM simulator used for all experiments carried out and elaborated in this section. We list its features and properties along with its limitations.
Then, we detail the implementation of the slot shifting algorithm in two main parts: In the first part, we describe the implementation of the offline phase of slot shifting. This includes the description of a tool that automatically generates the input file for the actual offline phase which then creates the annotated offline scheduling table. In the second part, we describe the implementation of the online phase of slot shifting.
After that, there follows the explanation of the experiments that we performed on the MPARM. The experiments cover a broad range of different system configurations as well as varying job set parameters. This allows us to analyze the runtime costs of different operations of the algorithm.
At the end of this chapter, we discuss the findings of our experiments. This also includes an analysis of the memory overheads of our implementation of slot shifting on the MPARM.

5.1 MPARM

MPARM is a cycle-accurate hardware simulation platform for multi-processor systems-on-chip (MP-SoC) which has been developed at the MicRel Lab, at the University of Bologna [71].

It allows modeling and connecting different hardware components using SystemC. The design paradigm of MPARM is based on modularization, which enables to integrate multiple instances of instruction set simulators, e.g., for the ARM architecture. MPARM includes models for memory and high-performance state-of-the-art buses, e.g., AMBA, STBUS, with different arbitration methods, e.g., TDMA, Round Robin (RR), TDMA+RR.

Further, the simulator comes with a C-library that offers a broad variety of support for "bare metal" operation without any underlying operating system. This includes:

- simulation control (functions to start and stop the collection of statistics, to obtain the current simulated time and to shut down simulated cores),
- atomic functions and synchronization primitives (test and set, signal, wait),
- inter-processor interrupts,
- basic functions for file handling, and
- macros that offer support for printing on the screen.

A GNU GCC based cross-compiler tool-chain to generate and run native ARM machine code in MPARM is available. Even complete operating systems like RTEMS and Linux for embedded systems (μCLinux) have been successfully ported to MPARM.

MPARM supports cycle-accurate simulation of ARM cores, which have been derived from the SWARM project [76]. SWARM's software model for the 32bit ARM architecture is entirely written in C++ and encapsulated in a SystemC wrapper to interface with MPARM. Figure 5.1 shows an overview of the SWARM module of a single core. Apart from the ARM core itself, the SWARM module also includes L1 data and instruction cache and some peripherals including an interrupt controller, a timer, and a local bus to connect them.

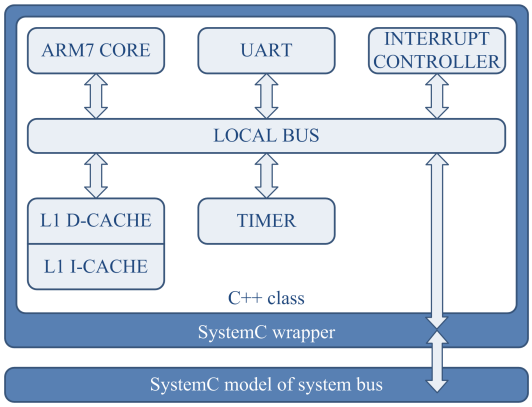


Figure 5.1: *SWARM module details.*

MPARM allows modifying the L1 cache configuration of the ARM cores, e.g., the cache can be unified or split into data and instruction cache. In general, there is support for fully associative, 2-way up to 32-way set associative, or direct mapped L1 caches with cache sizes in the range from a few bytes up to one megabyte.

Command line parameters are used to specify the configuration of the platform, i.e., the number of simulated cores, cache configuration, bus type and arbitration scheme, etc. The simulations on MPARM can be run with different bus and memory wait states, e.g., initial value, back-to-back, etc. MPARM can for each core log detailed statistics down to the level of each single bus/memory access. Slot shifting is not a very memory intense algorithm, as it only works on a limited small data set (see section 5.2.2.3), thus we consider experimenting with the memory specific settings not as the focus of our work.

In the standard setting that we use, the ARM cores run at 200MHz and feature an 8kB 4-way set associative L1 data cache, 8kB direct mapped L1 instruction cache, and have access to 1MB of core-private memory and 1MB of shared memory. For all measurements that we conduct, the bus is simulated using a transaction level based model (TLM) and the bus arbitration is set to TDMA+RR with a slot size of 50 nanoseconds.

5.1.1 Limitations

MPARM suffers from some limitations: First, there is no mechanism to pause a simulation once it has been started. Thus, a simulation cannot be run up to some synchro-

nization point, paused, and resumed when some external event occurs. Without such a feature, simulations run until they finish, or until they are forcefully terminated.

Second, MPARM offers no means to save the current state of the simulated environment to a file. If this were available, it would allow to store a simulation and to later resume from this predefined state arbitrarily often.

Third, MPARM encapsulates the simulated environment and thus renders it virtually inaccessible to the outside. While this at first is advantageous and required to perform undistorted simulations, it also forbids access to the internal data. Additionally, this encapsulation effectively blocks all attempts to trigger any event or action inside. Moreover, MPARM features no interface to actively access data from outside or to modify the simulated environment.
However, there exist some exceptions: When the simulations start, initial parameters can be sent to the simulated environment via the command line.
While it is infeasible to interact with the simulation from outside, the other way around is better supported. Code running within the simulation can utilize the pr-macros to print data to the screen or to a file. When printing to the screen, core IDs can be added to identify where the data originates from. When printing to file, this data is separately buffered for each simulated core and then stored into a core-specific file which can be read from outside MPARM. In the latter case, these pr-macros suffer from delays due to the buffering by MPARM and the underlying operating system that manages the file access.
Finally, MPARM offers rudimentary support to access files outside the simulated environment. Code executing on a simulated core can actively read data from a file. Thus, it is possible to trigger events inside MPARM during a simulation if the simulated code frequently polls on the content of a file.

5.2 Implementation of Slot Shifting on MPARM

As laid out in chapter 2, slot shifting consists of two parts: an offline phase, before system start-up, and an online phase, carried out at runtime of the system. Before we describe in the next sections our implementation of the offline and the corresponding online phase, we discuss a main challenge that influenced the design of both phases:

> *How to trigger the arrival of jobs of aperiodic tasks on the simulated ARM cores?*

The chosen approach must fulfill three requirements: First, these aperiodic jobs must be triggered *reliably*. If a job should arrive in the k-th slot, then the chosen approach must ensure that this job is not triggered too early or too late and thus arrives in a different slot. Second, aperiodic jobs must be triggered in a *reproducible* fashion. The complete description of an experiment not only consists of an offline task set and its parameters but also of the aperiodic tasks with their parameters, i.e., all parameters of their jobs. In other words, for an experiment description to be reproducible, it must also list fixed arrival times for the jobs of aperiodic tasks. The trigger mechanism must trigger the

arrival of the aperiodic jobs such that when the experiment is repeated, exactly the same results can be obtained again. Third, the chosen solution must be *suitable for the MPARM* simulation engine. Particularly, the chosen approach must be implementable on the MPARM simulator in a simple and straight-forward way, without causing too much implementation or runtime overheads.

In general, time progresses much slower for the simulated ARM cores inside MPARM than the real-world time of the operating system outside MPARM. This fact in principle facilitates triggering events such as the job arrival of aperiodic tasks from outside MPARM. Nevertheless, there are some issues when trying to synchronize the simulated cores in MPARM to the outside world in order to trigger the arrival of aperiodic jobs. These issues are caused by limitations of the MPARM simulator, which we described in the previous section: There is no mechanism to pause the simulation, or to directly trigger an event inside the simulator. However, there is support for file access from inside MPARM to the outside file system. We decided to not make use of it as the additional polling overhead appeared unreasonable. Further, this overhead leads to non-reproducible simulation results as the performance of the standard Linux operating system outside MPARM is subject to random influence by hardware, software, and user interactions. In other words, seen from MPARM, the performance of the hard disk varies unpredictably.

As there is no direct interface to trigger jobs of aperiodic tasks from outside the MPARM simulator, we have to trigger them from within the MPARM simulator. The basic idea is to embed all required information about future job arrivals of aperiodic task and their properties into the offline scheduling table. This allows to mimic the "random" arrival of aperiodic jobs with specified parameters on any ARM core at predefined moments in time. By doing so, experiments can easily by created, analyzed, and changed outside the simulation. Further, this approach guarantees reproducibility, since all experiments can be repeated arbitrarily often.

MPARM runs as a single threaded application and thus does not make use of multiple processors present in today's computers. We modified the run script of MPARM to allow for multiple instances of MPARM to run in parallel on different processors to speed up the simulations. The following sections elaborate on our implementation of the offline and the online phase of the slot shifting algorithm for the MPARM simulator.

5.2.1 Offline Phase

Our slot shifting implementation supports offline guaranteed tasks and firm aperiodic tasks. In order to ease the creation of scheduling tables, we split the creation into two steps that we will describe in the following two paragraphs: In the first step, we use a tool to automatically generate the input for the actual offline phase. In the second step, we trigger the actual offline phase with the input obtained from the first step and create and annotate an offline scheduling table. The output of the offline phase consists of automatically generated C-code with an embedded scheduling table. This scheduling table lists the parameters of the jobs of the offline tasks (such as earliest start times, processor to run on, WCET, etc.) as well as future arrivals of jobs of aperiodic tasks

and their parameters, respectively. In the next step, scripts automatically compile this C-code representation of the scheduling table together with the rest of the source code to create the executable file for the online phase. This executable is then uploaded to the simulated ARM cores inside MPARM and run to perform the actual online phase of the experiment. Finally, the scripts collect the results of the experiment and process the measured simulation data.

5.2.1.1 Input File Creation

Slot shifting models all offline and aperiodic activities on a job basis. For our experiments described in section 5.3 and especially in section 6.2, we need an automatic, fast and computationally not too complex method to generate a large quantity of random job sets. Initially, we aimed at producing job sets with precedence- and deadline-constrained offline jobs and firm aperiodic jobs. Both types of jobs should feature a predefined total utilization within an allowed error margin. "The problem of finding an optimal assignment of tasks to processors subject to precedence constraints has been shown to be NP-hard" [77]. In other words, even if we were able to produce such jobs sets very quickly, the mapping to processors is not solvable by a polynomial runtime algorithm in the general case. Heuristic-based approaches to map such jobs to the processors are the only practically[1] viable alternative with increasing number of jobs and processors as well as table length. Since heuristic tend to discard many unsuccessful intermediate mappings, using them can cause potentially long runtimes of the mapping process and thus of the overall job set generation process. Another disadvantage is that in total the number of required job sets is further increased, as heuristics do not provide any success guarantee for finding a suitable mapping for all given job sets.

Our approach to avoid this problem is to weaken the requirements on the precedence constraints of jobs. Instead, we aim at producing jobs sets consisting of two types of jobs with the following characteristics: First, to create offline guarantee workload we employ deadline-constrained offline jobs, derived from periodic tasks with a pre-defined total utilization. Second, we create randomly arriving firm aperiodic jobs with pre-defined total execution requirement.

Throughout the rest of this thesis we will refer to this total execution requirement of the aperiodic jobs as *utilization* created by the aperiodic jobs. We define the aperiodic utilization as the sum of all the aperiodic job's WCETs divided by the length of the scheduling table. That is, $U_{aperiodic}$ is the fraction of time that the core will potentially spend to process the aperiodic jobs throughout the length of the scheduling table.

The basic idea of the tool we developed to create the job sets is to generate periodic task sets whose utilizations match the desired utilization. Then, based on this task set, the tool creates the individual jobs. Finally, it adds aperiodic jobs until the desired aperiodic utilization is reached. These steps are repeated on each core which avoids the described mapping problem and automatically yields jobs sets for an arbitrary number of cores.

[1] Complete enumeration of this multidimensional search space for a feasible mapping takes too long and other, e.g., ILP-based approaches [37, 78] suffer from combinatorial explosion.

As a first step, the tool takes one or multiple XML files found in a specified folder as input. Each XML file determines the general parameters for a set of input files to be generated for the succeeding offline phase of slot shifting. Table 5.1 gives a complete overview of all parameters specified in such an XML file. The XML file determines the

general	offline tasks	aperiodic tasks
number of job set files	target utilization	target utilization
number of processors	max allowed error	min/max WCET
min/max simulation length	min/max number of tasks	DLX factor
	min/max WCET	
	min/max period	

Table 5.1: *Overview of the parameters for the input file creation for the offline phase.*

number of desired job set files for the experiment and further general parameters: the number of processors in the experiment and the upper and lower bound on the length of the simulation, i.e., the length of the scheduling table in slots. For both offline and aperiodic tasks, a target utilization is defined. Further, the maximum allowed error as a percentage of the target utilization for the offline tasks is given. The tool will discard any created job set whose total offline utilization deviates by more than this maximum allowed error. The XML file lists upper and lower bounds on further parameters such as the number of offline tasks, WCET, and period of tasks. The *DLX factor* allows to set the deadline D_i of the created aperiodic jobs as a multiple of their worst case execution time C_i and is defined as follows:

$$DLX = D_i/C_i \qquad \forall \text{aperiodic jobs}$$

For each XML file processed, the tool creates a separate output folder with as many text files as defined in the XML file. Each output file completely specifies the offline and aperiodic jobs of a single simulation run. As already mentioned, all jobs have already been mapped to the individual cores. The precedence constraints of the offline jobs have been resolved and are expressed by their earliest start times and deadlines. Furthermore, the tool has added the jobs of the aperiodic tasks and defined all their properties, i.e., their job parameters as well as their arrival pattern. To summarize, an individual output file determines the length of a specific simulation in slots and the number of simulated processors. Further, it lists all the job details, i.e., for each individual offline job the job ID, the ID of the processor to execute on, the earliest start time, the worst case execution time, and the deadline. Similarly, the file specifies for each aperiodic job a unique job ID, the ID of the processor it arrives on, the arrival time, the worst case execution time, and the deadline.

The simplified pseudo-code to generate a job set is shown in Listing 5.1. The complete algorithm is wrapped by an outer loop (line 12-41): in each iteration a complete job set is created. The tool uses the following iterative approach to construct a feasible job set:

First, a random[2] length of the scheduling table for this job set is determined (line 14). Then, the algorithm enters an inner loop (lines 16-36) in which for each processor a set of offline tasks with specified utilization is created. To do so, the algorithm determines within on the specified upper and lower bound the random number of tasks on the processor. Next, it creates a set of random periods for the tasks, again within the specified limits. Then, the algorithm determines a set which hosts for each task on this processor the utilization value. We use the UUniFast algorithm [80] to ensure that first of all, the sum of these task utilizations matches the specified target utilization from the XML file and second, to ensure a uniform distribution of the individual task utilizations. The runtime complexity of the UUniFast algorithm is $O(n)$ with n being the number of tasks. Thus the algorithm's runtime scales linearly with the size of the task sets and with the number of processors.

In the next step, the algorithm calculates the WCET of the tasks by multiplying the period values with the utilization values (lines 26-27).

Note that the rounding of the UUniFast results to integer values causes errors to the resulting utilization of the task set. Further errors are introduced due to the granularity of the allowed values for the task parameters. Thus, at the end of the inner loop, the algorithm checks whether the resulting utilization of the tasks violates the error margin (line 29). If this is the case, then all tasks of this processor are discarded and all steps of the inner loop are repeated for this processor (line 30). Otherwise, the algorithm adds all created tasks of this processor to the task set (lines 32-35). These steps repeat for each processor, until suitable solutions for all processors have been found and a complete job set has been generated.

After the periodic task set has been generated, the algorithm creates each task's individual jobs based on the corresponding task's parameters (line 38). Finally, the algorithm iteratively adds uniformly distributed aperiodic jobs until the resulting aperiodic utilization fulfills the requirement (line 40).

[2]To produce random numbers of 32bit length, we use the Boost library's implementation of a pseudo-random number generator, the Mersenne Twister MT19937, which is, by far, the most widely used pseudo-random number generator [79].

Listing 5.1: *Simplified pseudo-code to generate offline job sets. Input variables: numberOfJobSets numberOfProcessors min/maxTableLength min/maxNumberOfTasks min/maxT per Task min/maxC per Task overallTaskSetUtilization allowedErrorInTaskSetU.*

```
1  for (i = 0; i < numberOfJobSets; i++)
2  {
3    tableLength := uniform random number within [minTableLength, maxTableLength];
4
5    for(processorID = 0; processorID < numberOfProcessors; processorID++)
6    {
7      determine random value: numberTasksOnProcessor \\
8      within [minNumberOfTasks, maxNumberOfTasks];
9
10     create set of size numberTasksOnProcessor with periods of tasks:
11       setT[k] := uniform random number within [minT, maxT]
12
13     create set of size numberTasksOnProcessor with task utilizations:
14       setU := UUniFast(numberTasksOnProcessor, overallTaskSetUtilization)
15
16     create set of size numberTasksOnProcessor with WCETs of tasks:
17      setC[k] := min(max(setU[k] * setT[k], minC), maxC)
18
19     if (resulting utilization of current task set violates allowedErrorInTaskSetU)
20       restart loop for this processor;
21     else
22       for all tasks
23       {
24         add new task(processorID, setT, setC) to taskSet[i]
25       }
26   }
27
28   jobSet(i) := createJobs(tableLength, taskSet[i]);
29
30   createAperiodicJobs(tableLength, setU);
31 }
```

5.2.1.2 Offline Table Creation

In this section, we describe how our implementation of the offline phase creates and annotates an offline scheduling table for the online phase of slot shifting.

First, the offline phase parses the job set file created in the previous step and stores the jobs ordered by processor and type. Note that all the following steps are performed for each processor individually.
In a first step to create the intervals, the list of offline jobs is sorted in latest-deadline-first fashion. Then, we create new intervals beginning with interval ID 0 at the end of the time line. Therefore, we select the job with the latest deadline, i.e., the first from the list of offline jobs. The deadline of this job marks the end of the current interval. We add this job and all other jobs on the same processor with the same deadline to the new interval by establishing job-to-interval and interval-to-job links using IDs[3]. We iterate over all jobs until all jobs have been assigned to their intervals. Then, we calculate the start of all intervals, beginning with interval 0, i.e., the last interval in time. The start of an interval is defined as maximum of the end of the previous interval and the earliest possible start time of a job that belongs to the interval under consideration. Now, we have created all intervals that host jobs.
In the next step, we create and add empty intervals in between existing intervals if necessary. Additionally, we insert empty intervals at the end and at the beginning of the scheduling table. By doing so, we ensure that: First, the scheduling table features the same length on all processors. Second, every moment in time is covered by an interval, i.e., all slots belong to a defined interval, thus there is no undefined state during the online phase.
Then, we correct the interval IDs to make them start with 0 at the beginning of the time line. During this iteration, we also calculate and assign the spare capacity value for every interval according to Equation 2.3 from chapter 2.
In the next step, we additionally create for both offline and aperiodic jobs separate data structures which serve the purpose to store their arrival order.

After calculating all this data for the offline table, the annotated offline scheduling table is written to the hard disk. The output format of the offline phase is C-code. This automatically created code defines the necessary data types and holds the scheduling table. Additionally to the above mentioned data, the offline scheduling table derives and defines the following general parameters: the length of a slot in cycles, the number of (simulated) processors for the experiment and the total length of scheduling table in slots.

Listing 5.2 shows the most relevant data types used in our implementation. Jobs are represented by the struct *tJob* and intervals by the struct *tInterval*.

The most important variables are listed in Listing 5.3: The offline table specifies the total number of all jobs per processor ($g\_number\_of\_jobs\_on\_core$) and more specifically the total number of offline jobs per processor ($g\_number\_of\_offline\_jobs$) and aperiodic jobs per processor ($g\_number\_of\_aperiodic\_jobs$). Additionally, it lists

[3]Note that the offline phase will store these IDs in a file, while it is one of the initial steps of the online phase to translate these IDs into pointers.

Listing 5.2: *Declaration of data types in offline scheduling table for intervals and jobs.*

```
1  struct tJob
2  {
3          // system wide unique ID
4          short int uid;
5          short unsigned core_id;
6
7          // job types:
8          // 1: offline guaranteed job
9          // 2: aperiodic job, not yet arrived
10         // 3: aperiodic job, already accepted
11         // 4: aperiodic job, rejected
12         // 99: job that finished execution
13         short int type;
14
15         unsigned arrival_time;
16         unsigned wcet;
17         unsigned remaining_execution_time;
18         unsigned deadline;
19
20         // pointer to interval job belongs to
21         struct tInterval* interval_ptr;
22 };
23
24
25 struct tInterval
26 {
27         // unique ID (per core), starts with 0 on all cores
28         short unsigned uid;
29         unsigned start, end;
30         short int spare_capacity;
31
32         // used at runtime to create double linked list
33         struct tInterval* prev;
34         struct tInterval* next;
35
36         short unsigned number_of_jobs;
37         // entries in this array contain job IDs
38         struct tJob* jobs[MAX_JOBS_PER_INTERVAL];
39 };
```

the maximum number of intervals per processor (*g_number_of_intervals_on_core*) and defines some constants, e.g., for the upper bound on the number of offline jobs per interval (not shown in the listing).
The offline scheduling table lists the parameters of all aperiodic and of all offline jobs

sorted in earliest-deadline-first fashion in the *g_job_array*.
Additionally, the table features for both types of jobs a list, the *g_offline_jobs_index* and the *g_aperiodic_jobs_index*, which are lists of IDs of the job which arrives next, i.e., the entries of these lists are sorted according to earliest start times of the jobs. Finally, the scheduling table features a list of all intervals for each core, the *g_interval_list*. The list of intervals is sorted by ascending interval IDs, i.e., in timely fashion.

Listing 5.3: *Declaration of variables in offline scheduling table.*

```
1  short unsigned g_number_of_jobs_on_core[MAX_CORES];
2  short unsigned g_number_of_intervals_on_core[MAX_CORES];
3
4  short unsigned g_number_of_offline_jobs[MAX_CORES];
5  short unsigned g_number_of_aperiodic_jobs[MAX_CORES];
6
7  // all jobs, sorted according to deadline
8  struct tJob g_job_array[MAX_CORES][MAX_TASKS_PER_CORE];
9
10 // all intervals
11 struct tInterval g_interval_list[MAX_CORES][MAX_INTERVALS_PER_CORE];
12
13 // specify arrival sequence of jobs
14 struct tTask* g_offline_jobs_index[MAX_CORES][MAX_OFFLINE_JOBS];
15 struct tTask* g_aperiodic_jobs_index[MAX_CORES][MAX_APERIODIC_JOBS];
```

5.2.2 Online Phase

This section describes our implementation of the online phase of slot shifting. In order to conduct our experiments on MPARM, scripts cross-compile the code that implements the online phase together with the code that embodies the offline scheduling table. Then, the resulting binary is uploaded to the ARM cores within the MPARM simulation engine.

We implemented and run the online phase of slot shifting on top of the bare ARM cores, i.e., without any underlying operating system. Using an operating system would only add an additional layer of abstraction causing additional overheads and runtime penalties. This design decision allows to evaluate the efficiency of the algorithm itself, since the slot shifting algorithm runs directly on the platform. Our implementation provides reproducible results, not influenced by any management or background activities or other jobs executed by the operating system.

We program a timer to periodically fire and thus determine the beginning of a slot. All operations associated with the actual online phase of slot shifting are performed within the interrupt handler of this timer. At the end of the execution of the interrupt handler, a function that represents the execution of a job is called.

5.2.2.1 Slot Length

The first question that this design raises is: *what is a meaningful slot length?* For a comparison, today's standard Linux' Completely Fair Scheduler (CFS) executes processes "for about 5 * (1+log(m)) milliseconds and no less than 2*(1+log(m)) milliseconds" [19], where m is the number of processors. This results in 5ms (at least 2ms) on a single core system and about 8ms (at least 3.2ms) on a 4 core system. Nevertheless, the CFS does not implement a real-time scheduling policy and the resulting time frames are not guaranteed. In fact, in Linux many interrupts may fire unpredictably at any time and their interrupt handlers thus do "steal" substantial amounts of time.

We conducted some initial experiments to determine a meaningful length of a slot on MPARM. A too short slot length inhibits adequate work to be performed, while a too long slot length does not bring any additional gain: The execution time of the slot shifting algorithm itself and its sub functions is only a small fraction of the slot. Larger slot sizes only increase the overall simulation times for the experiments. We found slot sizes of about 200,000 cycles, i.e., 1ms to be a good compromise, as we are mainly interested to keep reasonable simulation times. The focus of this implementation is to analyze the runtime behavior of the slot shifting algorithm. Further increases of the slot length do not affect the runtime of the sub functions of the algorithm that we are interested to analyze. When implementing a real system, depending on the speed of the processor and the nature and constraints of the application, adjusting the slot sizes might be required.

5.2.2.2 Structural Overview

After the code has been uploaded to the ARM cores, it first performs some basic sanity checks on the data provided by the offline scheduling table. Then, the code starts the initialization process: It creates a double linked list of intervals and initializes the interval-to-job and job-to-interval pointers on each core. In the next step, a list of ready jobs is created by scanning the job list on the core. After these operations have been performed, the global versions of slot shifting reserve and initialize a part of the shared memory. This memory is later on used to delegate aperiodic jobs from one core to another. Finally, the timer that determines the beginning of the slot is programmed and enabled.

Every time this timer fires, the corresponding interrupt handler is called. The handler performs all slot shifting functionality of the online phase; its simplified control flow can be seen in Figure 5.2. In a first step, this interrupt handler updates the current slot counter and if needed the current interval counter. Then, it checks for newly arrived aperiodic jobs. The global versions of slot shifting start checking for jobs delegated from other cores first, before they look for locally arriving jobs. Any new aperiodic job has to undergo an acceptance test and is integrated into the schedule if this test succeeds. This integration might require to split an interval, i.e., create a new interval, add the newly arrived job to it, insert this new interval into the double linked list of intervals, and update the original interval's properties. Whether splitting is required or not, the spare capacity values of the old interval and the interval hosting the newly arrived job

need updating, which in case of borrowing might affect multiple previous intervals. If the acceptance test fails, then the partitioned version of slot shifting will simply reject this job, while the global versions try to delegate it to another suitable core as described in section 4.
In the next step, the list of jobs that are ready is updated. After that, the next job to run is selected on earliest-deadline-first basis and its properties are updated to reflect its execution. If there is no job ready, or if the selected job will be executing outside its interval, e.g., because of borrowing, the corresponding interval's spare capacities are updated. Note that this update might also involve multiple predecessor intervals. Finally, the handler will finish and schedule the selected job. During all these steps, the interrupt handler uses the data prepared in the offline phase.

In order to log the start and the completion of the sub functions, a logging macro is inserted into the code. This macro itself causes some overhead as it needs to obtain the time stamps that mark end and start and has to send them over the system bus to the shared memory.

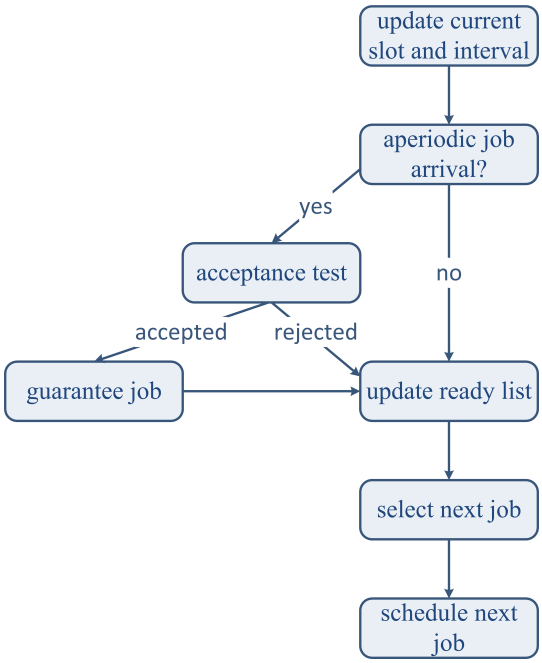


Figure 5.2: *Overview: simplified control flow of interrupt handler.*

5.2.2.3 Memory Overhead

Table 5.2 lists the memory requirements of our implementation of slot shifting on a 32-bit architecture. The table is divided into three parts: The upper part lists the memory needed to store the scheduling table, i.e., the overhead which is related to the number of aperiodic and offline guaranteed tasks. The middle part of the table lists the task independent memory overhead needed to perform slot shifting. The lower part

details the additional memory overheads for the global slot shifting algorithms and for logging of the overheads. With the given memory overhead, the following restrictions apply: The number of jobs and intervals per core is limited to 65,536 and the number of jobs per interval is restricted to 10. All job parameters, i.e., arrival time, WCET, and deadline, can range from 0 to 65,536 slots. Note that the global versions of slot shifting additionally require 12 bytes of shared memory per core and 22 bytes of shared memory for each single job that is delegated to another core. Since delegated aperiodic jobs can arrive to some core and can be delegated further in the same slot, the actual memory requirement to store the parameters of aperiodic jobs doubles to 44 bytes per job. The reason for this is that we decided to use static rather than dynamic data structures in the shared memory to optimize for speed.

Scheduling Table	
Variable Name	**Bytes**
job_list	$N_{total} * 22$
interval_list	$N_{total} * 58$
aperiodics_index	$N_o * 4$
periodics_index	$N_a * 4$
number_of_jobs	2
number_of_intervals	2
number_of_offline_jobs	2
number_of_aperiodic_jobs	2
Job Independent Overhead	
Variable Name	**Bytes**
ready_list	$N_{max\_ready} * 4$
additional variables	58
Shared Memory Overhead for Global Slot Shifting	
	Bytes
per delegated job	22
additional variables	13
Logging Overhead Per Core	
	Bytes
per timestamp	4
additional variables	12

Table 5.2: *Memory requirements of our slot shifting implementation per core, with N_{total}: total number of jobs, N_o: number of offline guaranteed jobs, N_a: number of aperiodic jobs, N_{max_ready}: max. number of concurrently ready jobs.*

5.3 Experiments on MPARM

To perform the experiments detailed in the following sections, we select the following MPARM settings: ARM cores are configured to have 1MB of private memory[4] and additionally 1MB of shared memory which is used for logging the runtime events and delegating aperiodic jobs.

The ARM cores run with the standard setting of MPARM: The L1 cache is split into a direct mapped instruction cache of 8kB and a 4-way set associative data cache that of 8kB. The system bus is simulated on transaction level and the arbitration method is set to two level: TDMA and Round Robin, with a TDMA slot size of 10 cycles (50 nanoseconds).

In general, we set the slot shifting slot size set to 200,000 cycles (i.e. 1 millisecond). The experiments are carried out on a Linux PC with an Intel Core i3 530 processor running at 2.93GHz. In order to measure the runtime of slot shifting and its sub functions, we embedded small logging macros into the code. These macros save timestamps and event IDs, which are evaluated at the end of the simulation.

5.3.1 Experiment 1: General Runtime Measurement for Partitioned Slot Shifting

In the first experiment, we measure the runtime of the online phase of the original slot shifting algorithm on a single core in four different scenarios:

1. no job execution (slots 0–25)
2. only offline guaranteed job execution (slots 25–50)
3. offline guaranteed and aperiodic job execution (slots 50–75)
4. only aperiodic job execution (slots 75–100)

The example schedule has a length of 100 slots and consists of 4 offline guaranteed jobs and 7 aperiodic jobs. Table 5.3 gives an overview of all jobs in this experiment and Table 5.4 presents the offline prepared disjoint intervals. The experiment also compares the costs of integrating 1, 2, or 3 aperiodic jobs at once into the schedule. The example has been created such that disjoint intervals must be split in order to guarantee jobs 5 – 10. Between slots 75 and 100, there is no job ready to execute and thus the core idles. At slot 90, a single aperiodic job arrives and is integrated into the schedule, which does not require splitting an interval.

Figure 5.3a shows the results from the runtime measurements, i.e., the time needed to perform the algorithm plotted over the 100 slots of the experiment. The figure lists the time spent to perform the sub functions (from bottom to top): to check for a new aperiodic job (in black), to perform the acceptance test and the guarantee algorithm

[4]The size of the binary that is uploaded to the private memory is usually less than 70kB, depending on the size and content of the scheduling table.

Job	Type	Earliest Start Time	WCET	Deadline
1	offline	25	10	50
2	offline	35	15	50
3	offline	50	10	80
4	offline	60	10	90
5	aperiodic	52	1	62
6	aperiodic	58	1	72
7	aperiodic	58	1	73
8	aperiodic	63	1	74
9	aperiodic	63	1	75
10	aperiodic	63	1	76
11	aperiodic	90	1	100

Table 5.3: *Job properties of Experiment 1.*

Interval	Start	End	Spare Capacity	Job IDs
1	0	25	25	–
2	25	50	0	1, 2
3	50	80	20	3
4	80	90	0	4
5	90	100	10	–

Table 5.4: *Experiment 1: offline generated intervals.*

(black/yellow shaded), to update the list of ready jobs (black/white fine hatched), to select the next job to be scheduled (dark gray), the interrupt service routine related overhead (white), and finally logging overhead (light gray).

In the very first slot, the runtime of the algorithm is comparably high, although all data structures have been initialized at system startup and although there is no activity in this slot. The reason for this is that the instruction and the data caches are still cold. The high cache miss rate in the first slot rapidly decreases in the second slot, as can be observed by the decreased overall runtime.

In the following slots, the measured runtime of the sub functions is constant except for minor fluctuations, as there exists no job in the first interval. In the slots 25, 35, 50, and 60 the runtime shows small peaks. These are the slots when the jobs of the offline guaranteed tasks become ready as planed in the offline table, so there is more time spent in these slots to update the list of ready jobs. Furthermore, small runtime peaks occur also in the slots 53, 59, 64-66, and 91. In these slots, offline guaranteed or aperiodic jobs finish their execution and thus more time is spent for updating the list of ready jobs.

Much higher peaks in runtime can be observed when the aperiodic jobs arrive. The acceptance test and the guarantee algorithm add significant overhead, especially if splitting is required, as Table 5.5 shows. The table includes the time needed to update the list of ready jobs, the time to schedule the next job (which includes time spend to update

all scheduling related variables and the maintenance of the spare capacities as well), the logging overhead, the overhead related to the interrupt service routine, and the total runtime. Adding a single aperiodic job leads to a total runtime of 47,545ns (17,745ns for the guarantee algorithm), adding two to 68,015ns (29,235ns), and three aperiodic jobs to 101,905ns (48,705ns). The runtime of the guarantee algorithm is approximately linear to the number of aperiodic jobs arriving in parallel. Figure 5.3a shows that when aperiodic jobs arrive, updating the list of ready jobs consumes more time. With the number of aperiodic jobs arriving in parallel, the runtime of this function increases.

Sub Function	**Slot**			
	52	**58**	**63**	**90**
check for new aperiodic job	4,015	5,440	7,250	2,145
acceptance test	7,130	8,600	13,795	3,955
guarantee algorithm	17,745	29,235	48,705	9,635
- add job	0	0	0	2,265
- add job & split interval	5,445	6,825	9,525	0
- update spare capacities	6,985	14,260	27,100	3,045
update ready list	7,735	10,965	14,440	3,690
next job selection	1,735	2,120	2,125	1,735
ISR overhead	2,905	2,790	2,790	2,730
logging overhead/misc.	6,280	8,865	12,800	3,120
total	47,545	68,015	101,905	27,010

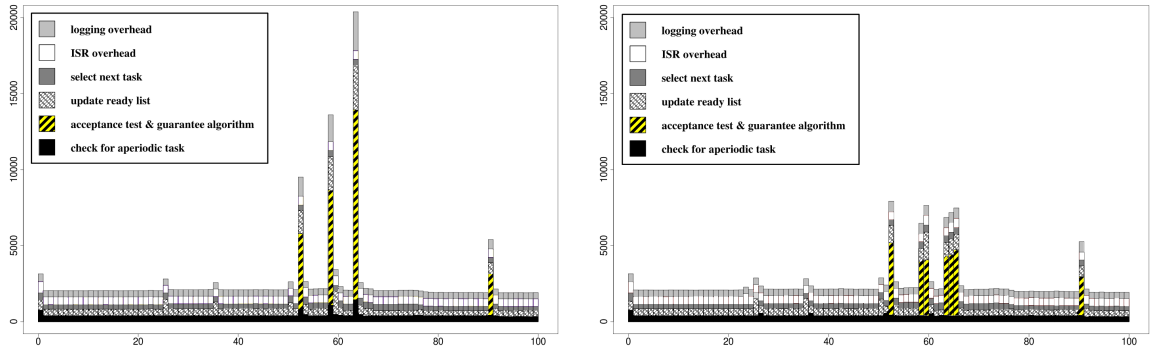
Table 5.5: *Runtime details of Experiment 1a (values in nanoseconds): multiple acceptance tests per slot.*

For comparison, Figure 5.3b shows results of Experiment 1b, in which we performed measurements with the same job set, but allowed only for *a single acceptance test per slot*. In this experiment, the acceptance test of all other aperiodic jobs is delayed for the next slot. This way, the runtime overhead is limited and still all jobs can be integrated into the schedule.
Table 5.6 details the measured runtime values for the distinct sub functions when only a single acceptance test per slot takes place. Since this approach leads to a simpler control logic, the total runtime in slot 52 decreased from 47,545ns to 39,625ns. In general, the total runtime is limited to less than 40,000ns for all slots with arriving aperiodic jobs.

The result of Experiment 1 also shows that the cost of adding an aperiodic job significantly decreases, if there is no need to split an interval. This can be clearly seen by comparing the runtime for integrating the first aperiodic job with the last aperiodic job of the experiment.

Sub Function	Slot						
	52	**58**	**59**	**63**	**64**	**65**	**90**
check for new aperiodic job	2,220	2,055	1,940	2,105	1,940	1,940	2,135
acceptance test	7,080	4,670	4,430	4,700	4,855	5,115	4,275
guarantee algorithm	16,645	12,780	13,990	14,005	15,255	16,600	8,460
- add job	0	0	0	0	0	0	2,325
- add job & split interval	5,370	3,470	3,265	3,360	3,245	3,245	0
- update spare capacities	6,615	6,095	7,510	7,435	8,800	10,145	2,655
update ready list	5,755	4,590	9,115	5,210	5,125	5,110	3,500
next job selection	1,850	2,170	2,350	2,305	2,245	2,185	1,955
ISR overhead	2,570	2,560	3,120	2,565	3,060	3,065	2,565
logging overhead/misc.	3,505	3,525	3,280	3,395	3,395	3,395	3,400
total	39,625	32,350	38,225	34,285	35,875	37,410	26,290

Table 5.6: *Runtime details of Experiment 1b (values in nanoseconds): single acceptance test per slot.*



(a) *Multiple Acceptance Tests per Slot.* **(b)** *One Acceptance Test per Slot.*

Figure 5.3: *Experiment 1: runtime of partitioned slot shifting, runtime values presented in cycles (1 cycle = 5ns).*

5.3.2 Experiment 2: Runtime Overhead of Splitting Intervals

The second experiment aims to quantify how much overhead is involved with the splitting of the intervals. The example schedule has a length of 100 slots and consists of 10 jobs of offline guaranteed tasks and 4 jobs of aperiodic tasks, whose properties are listed in Table 5.7. Table 5.8 shows the offline generated disjoint intervals and their properties. The deadlines of the aperiodic jobs do not coincide with the end of any offline created interval. The schedule is constructed such that each arrival of an aperiodic job triggers the splitting of an interval. Hence, a new interval needs to be inserted and the spare capacity values of the concerned intervals need to be updated.

Job	Type	Earliest Start Time	WCET	Deadline
1	offline	0	10	25
2	offline	40	5	60
3	offline	41	5	60
4	offline	42	2	60
5	offline	60	5	80
6	offline	61	5	80
7	offline	62	5	80
8	offline	50	5	100
9	offline	51	5	100
10	offline	52	5	100
11	aperiodic	5	2	50
12	aperiodic	23	2	70
13	aperiodic	38	2	75
14	aperiodic	69	2	90

Table 5.7: *Job properties of Experiment 2a and 3.*

Interval	Start	End	Spare Capacity	Task IDs
1	0	25	15	1
2	25	40	15	–
3	40	60	5	2, 3, 4
4	60	80	5	5, 6, 7
5	80	100	5	8, 9, 10

Table 5.8: *Offline generated intervals of Experiment 2.*

Figure 5.4 shows the time spend for different parts of the algorithm, similar to the Figures 5.3a and 5.3b from the previous experiment. Table 5.9 details the time that is spent in the different sub functions at these moments when aperiodic jobs arrive. The table lists the time spend to check for new aperiodic jobs, to perform the acceptance test, and if the test succeeded the time to guarantee the job. The time spend inside the

guarantee algorithm can be further subdivided in the time needed to either add the job to an already existing interval or the time needed to split an already existing interval, create an new interval and to add the aperiodic job, and finally to update the spare capacities of the intervals.

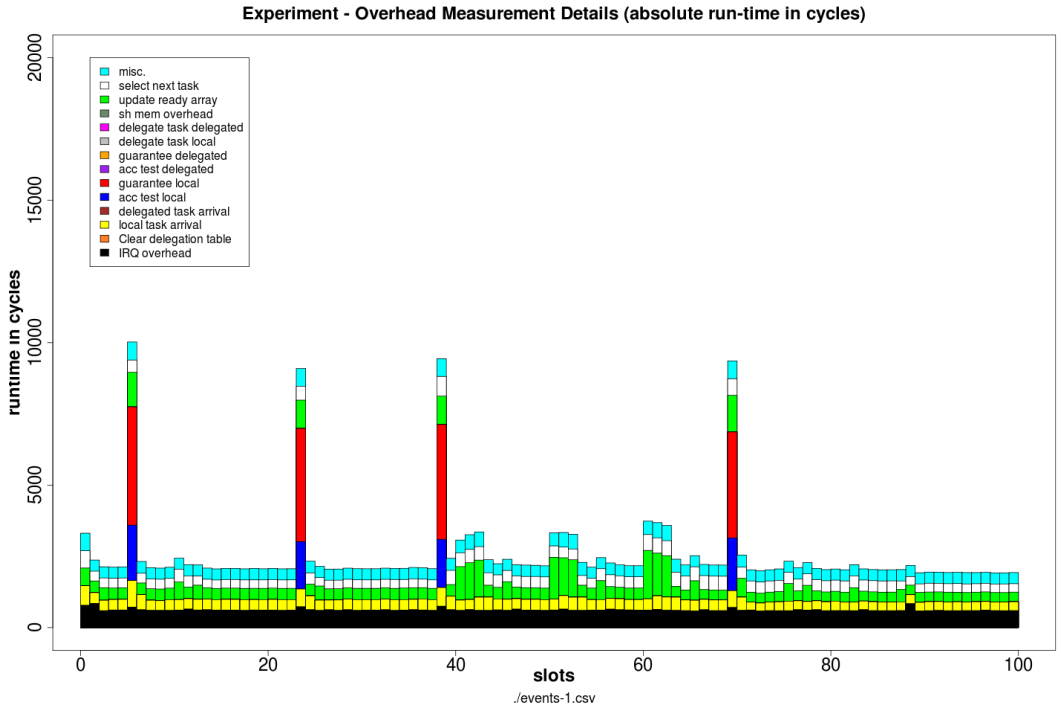


Figure 5.4: *Experiment 2a: Runtime of unmodified slot shifting, runtime values presented in cycles (1 cycle = 5ns).*

When an aperiodic job arrives, approximately half of the complete runtime of the slot shifting algorithm is required to perform the acceptance test plus the guarantee algorithm—most of it for the latter. Table 5.9 reveals that much of the time spent in the guarantee algorithm is required to split the interval and add the new job. Interestingly, the algorithm spends even more time in updating the spare capacities of the intervals. This offers much potential for improvement of the runtime if splitting can be avoided.

To confirm this presumption, we conduct another experiment: For Experiment 2b, we modify the job set such that the deadlines of the aperiodic jobs coincide with the ends of the offline derived intervals, see Table 5.10. Figure 5.5a shows the resulting runtimes and Table 5.11 lists the runtime details. The runtime required to perform the guarantee algorithm for the aperiodic job decreases by 5,400ns (6,730ns, 7,680ns, 7,920ns, respectively). As can be seen in the table, this is caused by two reasons: First, adding a job to an already existing interval is performed about twice as fast than splitting an interval into two and adding the job to a new interval. Second, when the

Sub Function	Slot			
	5	**23**	**38**	**69**
check for new aperiodic job	4,680	3,120	3,280	2,960
acceptance test	9,710	8,310	8,465	9,250
guarantee algorithm	20,745	19,865	20,150	18,600
- add job	0	0	0	0
- add job & split interval	4,620	3,745	3,965	3,915
- update spare capacities	10,665	12,160	12,225	10,620
update ready list	6,070	4,975	4,975	6,440
next job selection	2,150	2,420	3,450	2,895
ISR overhead	4,135	4,405	5,530	4,750
logging overhead/misc.	2,630	2,360	1,330	1,890
total	50,120	45,455	47,180	46,785

Table 5.9: *Runtime details of Experiment 2a (values in nanoseconds) – unmodified slot shifting.*

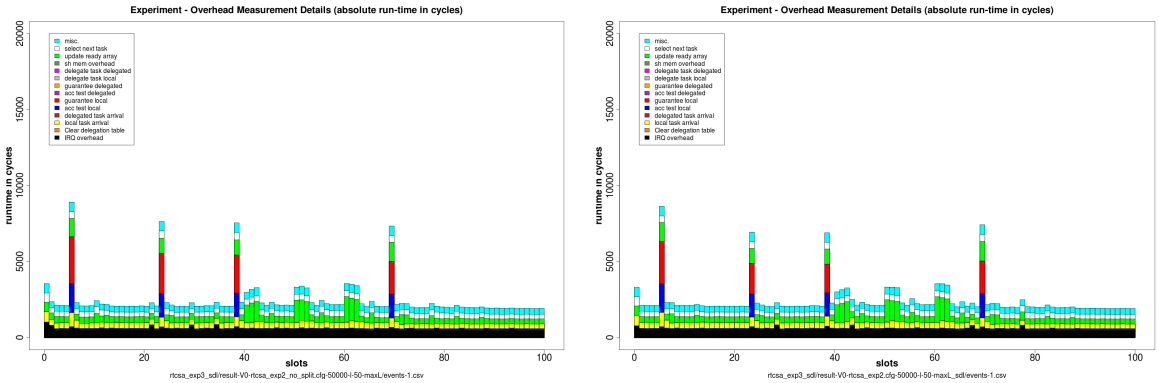
splitting is avoided, the update of the spare capacity values is much less costly in terms of runtime.

Encouraged by these findings, we modified the slot shifting algorithm, i.e., the acceptance test and the guarantee algorithm, as described in section 2.3.3. This improved slot shifting algorithm shortens the deadlines of aperiodic jobs as much as possible such that they coincide with the ends of already existing intervals. We refer to this new feature of the slot shifting algorithm as *SDL* (**S**horten **D**ead**L**ine, see page 32). For Experiment 2c, we employ SDL based on the previous (unmodified) job set, whose properties are listed in Table 5.7. Figure 5.5b shows the resulting runtimes and Table 5.12 lists the runtime details. As can be seen, our implementation of SDL yields similar results as the previous simulation with the modified job set. Table 5.13 lists the resulting relative runtime improvements compared to standard slot shifting. The results show that using SDL, up to 63.0% of the runtime of the guarantee algorithm and up to 26.9% of the overall runtime can be saved.

To summarize, SDL substantially improves the runtime requirement in (most) slots in which aperiodic jobs arrive: Since SDL aims to avoid costly interval splits and successive updates of their spare capacities, the time required to integrate aperiodic jobs into the scheduling table is reduced.

Job	Type	Earliest Start Time	WCET	Deadline
1	offline	0	10	25
2	offline	40	5	60
3	offline	41	5	60
4	offline	42	2	60
5	offline	60	5	80
6	offline	61	5	80
7	offline	62	5	80
8	offline	50	5	100
9	offline	51	5	100
10	offline	52	5	100
11	aperiodic	5	2	40
12	aperiodic	23	2	60
13	aperiodic	38	2	80
14	aperiodic	69	2	100

Table 5.10: *Job properties of Experiment 2b.*



(a) *Runtime Experiment 2b.* **(b)** *Runtime Experiment 2c.*

Figure 5.5: *Experiment 2b/c: runtime of modified job set vs. runtime of improved slot shifting algorithm using SDL, runtime values presented in cycles (1 cycle = 5ns).*

Sub Function	Slot			
	5	23	38	69
check for new aperiodic job	4,605	3,120	3,230	2,965
acceptance test	9,570	7,900	7,845	7,925
guarantee algorithm	15,345	13,135	12,470	10,680
- add job	2,650	1,665	1,650	1,705
- add job & split interval	0	0	0	0
- update spare capacities	7,155	7,555	6,795	4,890
update ready list	6,095	4,975	4,980	6,345
next job selection	2,165	2,415	2,465	2,205
ISR overhead	3,630	3,575	3,630	3,465
logging overhead/misc.	3,175	3,115	3,125	3,100
total	44,585	38,235	37,745	36,705

Table 5.11: *Runtime details of Experiment 2b (values in nanoseconds) – unmodified slot shifting, modified job set.*

Sub Function	Slot			
	5	23	38	69
check for new aperiodic job	4,670	3,115	3,275	2,915
acceptance test	9,465	7,690	7,745	8,025
guarantee algorithm	13,865	9,955	9,390	10,645
- add job	2,655	1,655	1,705	1,705
- add job & split interval	0	0	0	0
- update spare capacities	5,685	4,325	3,720	4,895
update ready list	6,260	4,980	4,975	6,500
next job selection	2,160	2,170	2,215	2,270
ISR overhead	3,620	3,630	3,775	3,580
logging overhead/misc.	3,170	3,120	3,115	3,170
total	43,210	34,660	34,490	37,105

Table 5.12: *Runtime details of Experiment 2c (values in nanoseconds) – improved slot shifting algorithm (SDL).*

Runtime Savings	Slot			
	5	23	38	69
absolute (in ns)	6,910	10,795	12,690	9,680
in % of runtime of guarantee algorithm	33.3%	54.3%	63.0%	52.0%
in % of total runtime	13.8%	23.7%	26.9%	20.7%

Table 5.13: *Experiment 2c: runtime savings compared to unmodified slot shifting algorithm.*

5.3.3 Experiment 3: Runtime Overhead on Multicore Systems for Partitioned Slot Shifting

For this experiment, we run slot shifting in a partitioned fashion on a simulated quad-core ARM machine to test the scalability of slot shifting on a multicore system. We employ the method previously described in section 5.2.1 to generate the offline table: We generate a random job set with uniformly distributed utilizations of the jobs of the offline guaranteed tasks by means of the UUniFast algorithm [80]. The rounding to integer values causes errors in the resulting total utilization of the task set. We select a task set for the simulation which features an error of less than 1% and run the offline phase to produce the offline scheduling table.

The parameters of the generated scheduling table are as follows: The length of the scheduling table on all cores is 500 slots and the target utilizations generated by the jobs of the offline tasks on each core is 50%. The earliest start times of all offline guaranteed tasks are set to zero and the deadlines are equal to the period. The WCET of the jobs are in the range of [1-15] slots and the deadlines are in the range of [15-30] slots.

The aperiodic jobs' arrival times are randomly generated between the start and the end of the scheduling table. Their WCETs are in the range of [10-15] slots, their deadlines are five times longer than the WCET of the job, and their total utilization is always equal to 50%, on each core.

In this scenario, we executed a total of 546 jobs on all cores; ordered by core: 117, 224, 25, and 98 offline jobs and 20, 21, 21, and 20 aperiodic jobs. To evaluate the results of all slots and all cores, Table 5.14 lists the obtained minimum, mean, and maximum *overall* runtimes in nanoseconds for the different parts of the algorithm. Note that the mean total runtime is smaller than the mean time to perform the guarantee algorithm. The reason for this is that in 1,918 out of in total 2,000 slots (on all cores) no aperiodic job arrives and thus the runtime is much shorter than in the 82 slots in which the guarantee algorithm is triggered because of aperiodic job arrivals (Table 5.15 lists the runtime details for slots *without* acceptance test).

Sub Function	Min. Runtime	Avg. Runtime	Max. Runtime
check for new aperiodic job	1,630	2,313.0	6,810
acceptance test	8,695	13,737.0	23,700
guarantee algorithm	15,040	30,472.0	46,655
- add job	2,645	2,941.0	4,990
- add job + split interval	1,665	2,190.5	2,640
- update spare capacities	8,155	23,690.5	40,350
update ready list	2,420	4,527.5	21,205
next job selection	1,600	2,624.0	2,725
ISR overhead	3,165	3,513.5	5,665
total runtime	11,320	17,130.5	93,525

Table 5.14: *Overall runtime details of Experiment 3 (in ns) – unmodified slot shifting.*

For this reason, Table 5.16 and 5.17 list the same values *only in those slots in which acceptance tests for aperiodic jobs take place*, for the unmodified and the improved version of slot shifting, respectively.

Sub Function	Min. Runtime	Avg. Runtime	Max. Runtime
check for new aperiodic job	1,630	2,187.6	3,710
update ready list	2,420	4,382.9	21,205
next job selection	1,600	2,603.1	5,115
ISR overhead	3,165	3,487.4	5,150
total runtime	11,320	15,194.3	32,625

Table 5.15: *Runtime details of Experiment 3 (in ns), only slots without acceptance tests – unmodified slot shifting.*

When comparing the total runtimes in all three categories for Table 5.15 and Table 5.16, we see that in the minimum (mean/maximum) case it took about 3 (4/3) times longer to integrate an aperiodic job into the schedule compared to perform pure table driven scheduling.

Sub Function	Min. Runtime	Avg. Runtime	Max. Runtime
check for new aperiodic job	4,565	5,245.5	6,810
acceptance test	8,695	13,737.0	23,700
guarantee algorithm	15,040	30,472.0	46,655
- add job	2,645	2,941.0	4,990
- add job + split interval	1,665	2,190.5	2,640
- update spare capacities	8,155	23,690.5	40,350
update ready list	3,730	7,908.0	15,360
next job selection	2,055	3,114.5	5,215
ISR overhead	3,810	4,130.0	5,665
total runtime	30,075	62,415.0	93,525

Table 5.16: *Runtime details of Experiment 3 (in ns), only slots with acceptance tests – unmodified slot shifting.*

When comparing the minimum total runtimes in Table 5.15 and Table 5.16, we see that the integration of an aperiodic job into the schedule increases the runtime by a factor of about 2.7. The same holds true for the mean and the maximum runtimes with an increase by factor of 4.1 and 2.9, respectively. In other words, accepting an aperiodic job adds a substantial amount of overhead compared to performing pure table driven scheduling.

Another interesting observation can be made: Table 5.16 reveals that there exist extreme cases in which updating the ready list—even when no new aperiodic jobs are present—can become a very costly operation (21,205ns!). This detrimental runtime behavior is caused by multiple coinciding events: Multiple offline guaranteed jobs became

ready to execute and at the same time one job has to be removed from the ready list. Furthermore, a negative effect on the runtime behavior can be observed, when new ready offline jobs feature very short deadlines, shorter than the deadline of any other ready job. In our implementation of the ready list, inserting a new job triggers a search starting from the end of the list, i.e., beginning with the ready job featuring the latest deadline. This implementation favors quick insertion of aperiodic jobs, since we assume new aperiodic jobs to have deadlines usually later than those of jobs already residing in the ready list. However, this implementation penalizes the insertion of new offline jobs with short deadlines.

Sub Function	Min. Runtime	Avg. Runtime	Max. Runtime
check for new aperiodic job	4,540	5,215.0	6,475
acceptance test	8,345	12,842.0	22,015
guarantee algorithm	10,470	19,378.0	35,455
- add job	2,815	4,309.0	4,900
- add job + split interval	1,595	1,725.5	2,785
- update spare capacities	4,980	13,595.0	29,760
update ready list	3,750	8,169.5	17,295
next job selection	2,040	2,597.0	3,955
ISR overhead	3,830	4,080.0	5,410
total runtime	27,795	52,259.5	87,245

Table 5.17: *Runtime details of Experiment 3 (in ns), only slots with acceptance tests – improved slot shifting (SDL).*

Table 5.18 presents the measured runtime savings when employing SDL. In the case of maximum runtime, the improved algorithm decreases the total runtime by 6,280ns, which is 6.7% of the total runtime.

Runtime Savings	Min. Runtime	Avg. Runtime	Max. Runtime
absolute (in ns)	2,295	10,155.5	6,280
in % runtime of guarantee algorithm	15.3%	33.3%	13.5%
in % of total runtime	7.6%	16.3%	6.7%

Table 5.18: *Experiment 3: runtime savings compared to unmodified slot shifting algorithm.*

In the case of minimum runtime, the improved algorithm saves approximately 8% of the total runtime, as the runtime is already very short (30,075ns) and the absolute time improvement is 2.295ns. On average, the improved algorithm saves 33.3% of the runtime of the guarantee algorithm, which results in a reduction of 16.3% of the total runtime of the algorithm (10,155.5ns).

The results of this experiment confirm the findings of the previous experiment. To summarize: SDL yields substantial runtime improvements in both unicore and multicore

scenarios. While these improvements are not as pronounced in the extreme cases of the observed minimum and maximum runtimes, SDL yields average runtime improvements of a third of the runtime of the guarantee algorithm.

5.3.4 Experiment 4: Runtime Overhead on Multicore Systems

The purpose of this experiment is get more insights into the runtime behavior of the different slot shifting algorithms. Hence, we determine the execution times of the original slot shifting algorithm and of our two global algorithms: algorithm 1, spare-capacity-based slot shifting, and algorithm 2, negotiation-based slot shifting. For this experiment, all measurements are carried out on the same simulated quadcore system with the same settings as used for the previous experiment. We observe the minimum and maximum execution times of the algorithms and we determine the average execution time with 95% confidence interval.

As in the previous experiment, we used the UUniFast algorithm [80] to prepare task sets with 30% offline task utilization per core and 50% aperiodic task utilization per core (see Table 5.19 for details). We then created and annotated offline scheduling tables of 500 slots for each of the four cores as described in section 5.2.1. After that, we performed the runtime measurement for the partitioned as well as for both global algorithms.

Parameter	Offline Jobs	Aperiodic Jobs
WCET	$[1, 15]$	$[10, 15]$
Period/Deadline	$[15, 30]$	$2 * WCET$
Resulting U per core	30%	50%

Table 5.19: *Experiment 4: overview of the job parameters.*

To obtain a better overview, we define three distinct categories to classify the measured execution times of each algorithm on a core in a slot:

A. slots in which no acceptance test takes place

B. slots in which acceptance tests for local aperiodic jobs take place; but no acceptance tests for delegated aperiodic jobs

C. slots in which acceptance tests for delegated aperiodic jobs take place (additionally local acceptance tests can take place)

To provide a general overview, Table 5.20 lists the resulting runtimes of the different algorithms for a job set consisting of 30 offline jobs and 34 aperiodic jobs per core, i.e., 256 jobs in total. The results are presented according to categories and the values are measured in nanoseconds. We present the observed minimum, average, and maximum runtime for each distinct category. As the partitioned slot shifting algorithm does not support migrations, there exist no slots that fall into category C. In general, both global algorithms require more execution time than the partitioned version. Furthermore, global algorithm 1 generally runs slower than global algorithm 2.

Exec. Time	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category A		
min.	9,560	19,475	15,795
avg.	11,355 ±59 (1,300)	24,289 ±97 (2,112)	19,298 ±77 (1,649)
max.	18,925	32,505	27,510
	Category B		
min.	23,380	43,545	32,905
avg.	39,442 ±1,212 (7,213)	57,993 ±1,107 (6,512)	47,783 ±1,316 (7,628)
max.	55,290	69,135	64,880
	Category C		
min.	-	42,015	29,740
avg.	-	61,264 ±3,532 (11,398)	42,362 ±2,388 (12,183)
max.	-	92,820	80,540
	Overall		
min.	9,560	19,475	15,795
avg.	13,265 ±325 (7,423)	27,270 ±446 (10,171)	22,288 ±401 (9,138)
max.	55,290	92,820	80,540

Table 5.20: *Exp. 4: overview of measured execution times (in ns) with 95% confidence interval, standard deviation in brackets.*

Additionally, we are interested in the detailed runtime behavior of all slot shifting algorithms. Figure 5.6 shows the average time that the different slot shifting algorithms spent in sub functions in slots in which acceptance tests took place, i.e., category B for partitioned slot shifting and categories B and C for the global algorithms. Notice that all the detailed results which were used to obtain the figure can be found in Appendix A. Tables A.1, A.2, A.3, A.4, and A.5 list all the measured runtimes for the different categories A, B, C, B + C combined, and the overall runtimes of all categories combined, respectively.

Figure 5.6 reveals that partitioned slot shifting spends approximately twice as much time to update the list of ready jobs as to check for newly arrived aperiodic jobs. Furthermore, partitioned slot shifting spends about one fourth of its complete runtime to perform the acceptance tests and nearly one third to guarantee aperiodic jobs. Only about 6% of the total runtime is required to decide which jobs to execute next and to perform the maintenance of the spare capacities. "*ISR overhead*" denotes the time that is spent in the interrupt service routine—in this algorithm 8.6% of the total runtime—for updating flags, saving and restoring the context, etc. The remaining time, about 8.4% of the total runtime, is spent to update counters (e.g. the slot and the interval counter), to make decisions whether to jump into sub functions or not, and for logging of events.

For both global slot shifting algorithms, a new category "*table overhead*" is introduced to represent the runtime needed to exchange data among the cores via shared

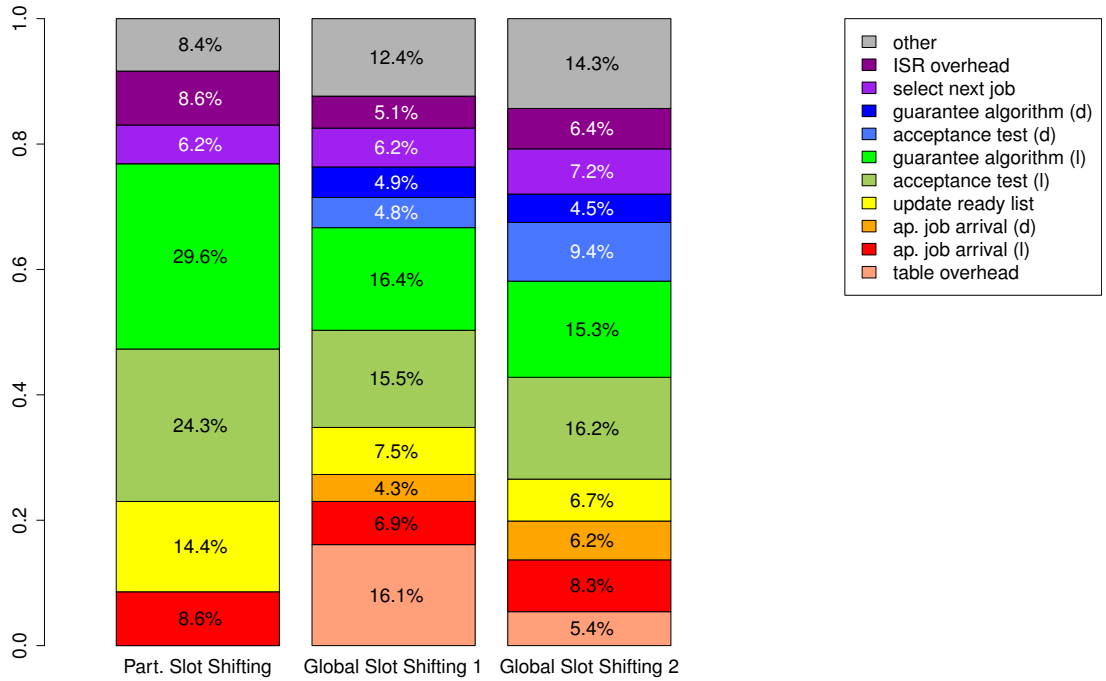


Figure 5.6: *Experiment 4: Normalized runtimes of partitioned and global slot shifting for slots of category B and C.*

memory. Evidently, the spare-capacity-based global slot shifting algorithm spends a substantial amount of time accessing and maintaining information via shared memory—approximately three times more than the other global algorithm.

Furthermore, Figure 5.6 shows how much time is spent to check for new aperiodic jobs and how much time is spent when acceptance test and guarantee algorithm are invoked for a local aperiodic job (l) or for an aperiodic job that has been delegated from another core (d).

Interesting to note is that, although both global algorithms run based on the same scheduling tables, their behavior differs in many details: Global algorithm 1 manages to accept 121 of the 136 aperiodic jobs locally[5]. As a result, in total 40 delegations for the remaining 15 jobs take place. Out of the 15 delegated jobs, 11 jobs can be successfully integrated, resulting in 132 accepted jobs in total.
Global algorithm 2 only accepts 102 of the 136 aperiodic jobs locally and delegates the rest to other cores. This results in 100 delegation events on the four cores, i.e., in 100 slots acceptance tests for delegated aperiodic jobs from other cores take place. Out of the 34 delegated jobs, global algorithm 2 accepts 31 additional aperiodic jobs, leading to a result of 133 accepted aperiodic jobs in total.

Nevertheless, global slot shifting 1 spends 16.1% of the runtime to share data among cores, whereas global algorithm 2 only spends 5.4% of the runtime for data exchange.

[5]For comparison: In this experiment, partitioned slot shifting accepts 112 aperiodic jobs.

The explanation is that the former algorithm heavily depends on the available spare capacities of the other cores: In global algorithm 1, all cores share their available spare capacities with all other cores. They frequently check the other cores' values to make a decision to which core to delegate aperiodic jobs to if the local acceptance test fails.

When local acceptance fails, global slot shifting 2 delegates the corresponding aperiodic job to the remaining cores to perform the acceptance test. As one would expect, Figure 5.6 shows that global slot shifting 2 spends more time to perform acceptance tests for delegated jobs than global slot shifting 1. Once a suitable core to integrate an aperiodic job is found, both algorithms invoke the same guarantee algorithm. Thus, the time spent to perform this operation is similar in both global algorithms. Notice that although three other cores are potential candidates if a local test fails, the time that global slot shifting 2 spends for acceptance tests of delegated jobs does not triple. Instead, the increase is only by a factor of 1.96, which shows the efficiency of the implementation.

Multiple interesting conclusions can be drawn: First, given a total system utilization of 80% per core, global algorithm 1 delegates locally rejected aperiodic jobs quite often, i.e., 2.7 times on average, before they are accepted or finally rejected. Second, in exactly the same scenario global algorithm 2 performs more delegations: 2.94 delegations on average. Third, although global algorithm 2 leads to much more delegation events, its resulting runtime is much lower (see results in Table 5.20 and tables in Appendix A for absolute values).

While partitioned slot shifting is less flexible in the handling of aperiodic jobs, since jobs are only handled locally, it offers the lowest runtime requirements. In contrast to this, both other algorithms can handle aperiodic jobs globally, at the cost of speed. The advantage of global slot shifting 1 over global slot shifting 2 is that less time is spent for acceptance tests of delegated aperiodic jobs. In order to achieve this, global algorithm 1 heavily relies on knowledge of available spare capacities. Hence, approximately three times more time is necessary to share data among the cores (see table overhead). As shown in Table 5.20, in total this results in global algorithm 2 being faster compared to the other global algorithm. Finally, using the data from Appendix A, Table 5.21 presents the additional runtime requirement of the global slot shifting algorithms compared to partitioned slot shifting for different slot categories.

Category	Global Algorithm 1	Global Algorithm 2
Category A	+ 113.9%	+ 70.0%
Category B + C	+ 48.0%	+ 15.1%
Overall	+ 105.6%	+ 68.0%

Table 5.21: *Experiment 4: Additional runtime of global slot shifting algorithms compared to partitioned slot shifting.*

5.3.5 Experiment 5: Impact of the Number of Jobs on the Runtime Requirements

The aim of this experiment is to evaluate the impact of the job set size, i.e., the total number of jobs, on the runtime of the slot shifting algorithms. The set-up is the same as in Experiment 4 and the task set parameters are identical to those presented in Table 5.19. This experiment consists of two independent scenarios: Experiment 5a uses a similar job set as in the previous experiment, but the 30 offline jobs per core are exchanged against 70 offline jobs per core from other randomly generated job sets. Experiment 5b is also based on the job set of Experiment 4, but here the 30 aperiodic jobs on each core are replaced against 80 aperiodic jobs per core.

Table 5.22 lists the additional runtimes of Experiment 5a[6] compared to Experiment 4. As expected, increasing the number of the offline jobs from 30 to 70 results in an increase of the average overall runtime of all slot shifting algorithms. The average overall runtime of the partitioned slot shifting algorithm increases by 15%, and the runtime of the global algorithms 1 and 2 increases by less than 10% (3%) percent, respectively. Obviously, for this job set, although it consists of more offline jobs, the worst case runtime behavior is sometimes better than that of the previous job set used in Experiment 4. Therefore, the maximum runtime drops in category B for partitioned slot shifting and in all categories for global slot shifting 2. Furthermore for global slot shifting 2, the minimum overall runtime as well as the runtime in slots when there is no acceptance test (category A) decreases.

Category	Additional Runtime in % (Minimum/Average/Maximum)		
	Orig. Part. Slot Shifting	**Global Algorithm 1**	**Global Algorithm 2**
A	12.0/16.6/10.7	1.0/10.1/16.6	-1.4/2.0/-1.4
B	5.5/8.8/-2.2	8.8/7.9/6.6	4.7/5.2/-3.6
C	- / - / -	7.3/3.1/3.1	0.0/5.7/-9.8
Overall	**12.0/15.0/-2.2**	**1.0/9.8/3.1**	**-1.4/2.8/-9.8**

Table 5.22: *Experiment 5a: performance with more offline jobs.*

Table 5.23 lists the additional runtimes of Experiment 5b[7] compared to Experiment 4. As expected, when the number of aperiodic jobs per core increases from 30 to 80, the average overall runtime of the slot shifting algorithms increases as well. The average overall runtime of the partitioned slot shifting algorithm increases by 37.9%, and the runtime of the global algorithms increases by about 15%. Note that the additional overall runtime is higher than in each of the distinct categories A, B, or C. The reason for this discrepancy is that although slots of category B and C occur more rarely, their measured runtimes are much higher than those of slots of category A. In other words,

[6]The detailed results of the experiment can be found in Table B.1 in Appendix B.
[7]The detailed results of the experiment can be found in Table B.7 in Appendix B.

a small relative increase in slots of categories B or C leads to this bigger increase of the overall runtime.

Category	Additional Runtime in % (Minimum/Average/Maximum)		
	Orig. Part. Slot Shifting	**Global Algorithm 1**	**Global Algorithm 2**
A	16.0/18.4/19.1	5.8/3.6/8.9	-1.5/2.0/9.8
B	4.2/8.2/-2.7	-2.4/2.9/4.7	-1.5/4.2/-4.3
C	- / - / -	5.5/4.2/4.1	-2.9/2.3/9.6
Overall	**16.0/37.9/-2.7**	**5.8/15.4/4.0**	**-1.5/14.7/9.6**

Table 5.23: *Experiment 5b: performance with more aperiodic jobs.*

To summarize, adding more jobs to the job set typically increases the runtime as more job/interval parameters need to be maintained. The experiment shows that the runtime of all slot shifting algorithms increases only by a much smaller factor than the job set size. Further, the impact of increasing the number of offline jobs per core on the average runtime is smaller than the impact of increasing the number of aperiodic jobs. The reason for that is that every additional aperiodic job triggers at least one acceptance test and in case of success the guarantee algorithm. And as Experiment 4 has previously shown, these functions that integrate aperiodic jobs are very time intensive (see Figure 5.6).

Note that the detailed results that Table 5.22 and 5.23 are based on are listed in Appendix B. For comparison reasons, similar figures to Figure 5.6 from Experiment 4 can be found there as well.

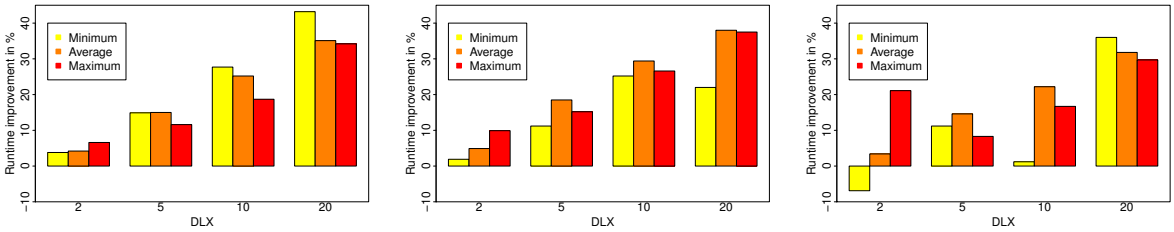
5.3.6 Experiment 6: Impact of the Deadline/WCET Ratio on the Runtime

The previous experiments presented the different costs that are associated with integrating aperiodic jobs into offline schedules. A substantial amount of time is spent to perform the acceptance test and the succeeding guarantee algorithm to add the job to the schedule and manage the interval and spare capacity information. In experiment 2, we presented a method to modify the deadline of accepted aperiodic jobs to improve the runtime of partitioned slot shifting. The method, called *SDL*, tries to smartly shorten the deadline of an accepted aperiodic job as much as the available spare capacities allow. An additional constraint is to place the deadline at the end of an already existing interval, to avoid the overhead of creating a new interval.

This experiment evaluates the runtime improvements for partitioned and global slot shifting using SDL. The parameters of the experiment are similar to those of experiment 4, i.e., the utilization created by the offline tasks is 30% and by the aperiodic tasks is 50%. For comparison reasons, the offline jobs are exactly the same as in experiment 4. In section 5.2.1.1, we introduced the *DLX* factor. In this experiment the DLX factor is varied to quantify the influence of the length of the deadline of the aperiodic jobs (with

respect to their WCET) on the runtime behavior of slot shifting when using SDL. Remember that changes to the DLX factor affect only the aperiodic jobs of the job set. For the aperiodic jobs of the job set, the deadline is set according to: $D_i = DLX * C_i \quad \forall i$. More precisely, in this experiment their deadlines are set to 2, 5, 10, and 20 times their worst case execution time.

Table 5.24 lists the measured average runtime improvements in slots with acceptance tests. Figure 5.7 shows the same data for each of the distinct slot shifting algorithms plotted as bar plots for the different DLX factors, respectively.



(a) *Partitioned slot shifting.* **(b)** *Global slot shifting 1.* **(c)** *Global slot shifting 2.*

Figure 5.7: *Experiment 6: min/avg/max runtime improvement for the three slot shifting algorithms.*

Independent of the observed algorithm, when using SDL the average runtimes improve (i.e., decrease) with increasing DLX factor. Using the partitioned slot shifting algorithm, the minimum and maximum runtimes improve with increasing DLX factor as well. The same holds true for the global slot shifting algorithm 1, with a small exception for DLX 20: the minimum runtime improvement drops slightly below the corresponding value at DLX 10. For the last algorithm, we see at DLX 2 a deterioration of 6.9% of the observed minimum runtime. Similarly, at DLX 10 the improvement of the minimal observed runtime by using SDL is only marginal. The explanation for this effect in

Runtime Improvement in %	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	DLX 2		
min/avg/max	3.8/4.2/6.6	1.9/4.9/9.9	-6.9/3.4/21.1
	DLX 5		
min/avg/max	14.9/15.0/11.6	11.2/18.5/15.2	11.2/14.6/8.3
	DLX 10		
min/avg/max	27.7/25.2/18.7	25.2/29.4/26.6	1.2/22.2/16.7
	DLX 20		
min/avg/max	43.2/35.1/34.2	22.0/38.0/37.5	36.0/31.8/29.7

Table 5.24: *Experiment 6: average runtime improvements in slots with acceptance tests (category B + C).*

this scenario is that by shortening the deadlines four of the aperiodic jobs cannot be accepted locally and get delegated, thus, algorithm 2 has to pay runtime penalty. This effect does not necessarily occur for all scheduling tables. To summarize: When SDL is applied, global slot shifting 1 shows the biggest improvements of the average runtime, followed by the partitioned and global slot shifting 2.

5.4 Discussion

In section 5.3, we presented the experiments that we conducted with the MPARM simulation engine. We thoroughly analyzed the efficiency of slot shifting, i.e., the overheads and the runtime costs associated with the partitioned and the two global slot shifting algorithms. Additionally, we modified the existing guarantee algorithm to modify the deadlines of aperiodic jobs (SDL) and evaluated the properties of the resulting algorithm. In this section, we summarize and discuss the most important findings and insights gained from the experiments.

Experiment 1 evaluates the runtime requirements of the partitioned slot shifting algorithm on a single core. In general, as long as no aperiodic jobs arrive, the observed runtimes are nearly constant in all slots. Minor fluctuations only occur when new offline jobs become ready or when offline jobs finish.
There exists one notable exception: In the first slot, the runtime requirement of slot shifting is significantly higher due to frequent initial cache misses, i.e., the cache is still cold. After the initial warm up of the cache during the first execution of the code, the effect vanishes in the succeeding slots. In our simulations we only schedule a small dummy job between the periodic executions of the slot shifting algorithm. This dummy job features only a small cache footprint, thus the cache stays hot and unaffected.
The activities of a real application running during a slot might evict the data of slot shifting from the cache. It is likely that a real-world implementation of slot shifting thus has to pay a similar runtime penalty every slot as we observed before the first slot.
Nevertheless, our implementation suffers additional runtime penalties in every slot due to the logging overheads that real implementations will be free of. Thus, we believe that both effects approximately cancel each other out.

The observed runtime requirements of the slot shifting algorithm are not constant anymore in slots in which aperiodic jobs arrives: Acceptance tests and the succeeding execution of the guarantee algorithm cause major spikes in the observed runtime. When multiple aperiodic jobs arrive in the same slot, the observed runtime of the guarantee algorithm increases linearly with their number.
To bound these observed spikes, we propose to allow only for a single acceptance test per slot. In case multiple aperiodic jobs arrive concurrently, their acceptance test is delayed for the next slot[8].

[8]To allow for a meaningful support of global algorithms, aperiodic jobs must anyway feature some leeway. Thus, the impact of this small delay on the acceptance ratio of the algorithm is negligible.

We saw already in the first experiment, that a substantial amount of time is required to integrate an aperiodic job. Especially when an existing interval needs to be split, a substantial amount of the total runtime is required to perform the corresponding operations: to add the new aperiodic job to the newly created first interval, to move all jobs from the original interval to the newly created second interval and to update the spare capacity values of the affected intervals. To overcome this issue, we propose an improved version of the guarantee algorithm of slot shifting, named *SDL*. This method aims at shortening the deadlines of the aperiodic job as much as possible, such that it coincides with an already existing interval. Thus, the costly operations mentioned before can be avoided. Experiment 2 shows that applying SDL leads to lower runtime requirements in most slots in which aperiodic jobs arrive. In the given unicore experiment, SDL is able to reduce the runtime of the guarantee algorithm by up to 63%.

In the succeeding Experiment 3, performed on a simulated quadcore system, SDL manages to save up to 33% of the runtime of the guarantee algorithm. The absolute runtime saving of SDL in both experiments lies in a magnitude of approximately 10,000 to 13,000 nanoseconds.
Other interesting observations can be made from the results of Experiment 3, concerning the costs of integrating aperiodic jobs into an offline scheduling table: When comparing the required runtime of the fastest slot in which no aperiodic job arrives with the fastest slot in which an aperiodic job arrives, we see that for the latter slot, the runtime requirements increase by an factor of 2.7. The average factor between all slots without and with arriving aperiodic jobs is 4.1; and the factor between these two slots without and with aperiodic job arrival which required most runtime is 2.9. In other words, there is a significant cost that is associated with the integration of aperiodic jobs by the slot shifting algorithm compared to purely table-driven scheduling.

Experiment 3 also reveals that—even when there are no aperiodic jobs present—an extreme case exists in which updating the ready list consumes nearly five times more time than in the average case. In this extreme case that occurred in one slot, the observed maximum runtime of the sub function that updates the ready list even exceeds the average runtime of the complete slot shifting algorithm in other slots[9]. Our analysis shows that this extreme case is triggered when multiple events coincide and thus their runtimes add up: In the same slot, a finished job has to be removed from the ready list and multiple offline jobs with comparably short deadlines are added to the ready list. Our implementation penalizes adding these offline jobs, since we optimized it for quick insertion of new aperiodic jobs (which usually feature comparably large deadlines).

The detailed analysis of Experiment 4 reveals those sub functions that dominate the overall runtime in slots in which acceptance tests for local aperiodic jobs take place: The partitioned slot shifting algorithm spends approximately 25% of the total runtime to perform the acceptance test, approximately 33% to run the guarantee algorithm, and approximately 15% of the runtime to update the ready list.

[9] In this experiment, the observed maximum runtime for updating the ready list is 21.2μs, while the average runtime of the complete slot shifting algorithm is 17.1μs.

Compared to the partitioned algorithm, both global algorithms require additional time to perform operations related to sharing information among the cores, the so called "table overhead". As the spare-capacity-based global algorithm 1 heavily relies on the exchange of all the cores' currently available spare capacity values, it features three times more table overhead than the other global algorithm. Apart from that, both global algorithms feature an approximately similar runtime distribution. The only noteworthy exception is that global algorithm 2 spends on average 1.5 times more time to perform acceptance tests on delegated jobs than global algorithm 1

In Experiment 4, global algorithm 1 delegates locally rejected aperiodic jobs quite often, i.e., 2.7 times on average, before they are accepted or finally rejected. However, although global algorithm 2 creates more delegation events than its counterpart, the performance results favor algorithm 2 in terms of runtime.
In short: In the given experiment, the partitioned slot shifting algorithm performs best, followed by the negotiation-based global algorithm 2 and then the spare-capacity-based global algorithm 1. The resulting average runtimes of the global algorithms are with 205.6% and 168.0% quite substantial[10]. In chapter 6, we will analyze and evaluate the effectiveness and resulting benefits of the algorithms in depth.

In Experiment 5, we show that—as one would expect—adding jobs to the job set increases the overall runtime, since more jobs and more intervals need to be managed. This runtime increment is much smaller than the factor by which the number of jobs of the job set is increased, e.g., increasing the number of offline jobs by a factor of 2.3 only yields a maximum runtime increase of 15.0%.
In general, adding aperiodic jobs has a much higher impact than increasing the number of offline jobs: Increasing the number of aperiodic jobs by a factor of 2.7 leads to a maximum runtime increase of 37.9%. This bigger impact of aperiodic jobs is to be expected: Since every aperiodic job additionally triggers at least one accept test and in case of success, the guarantee algorithm—which is a quite time consuming operation.

With the last experiment, we show the impact of the deadline to worst case execution time ratio (i.e., the DLX factor, as defined in section 5.2.1.1) of the aperiodic jobs on the runtime improvements that SDL yields for the different slot shifting algorithms: Global algorithm 1 gains most from SDL, followed by partitioned slot shifting and global algorithm 2. In general, the experiment confirms an intuitive expectation: SDL achieves more substantial runtime improvements for jobs sets with bigger deadline to worst case execution time ratio, i.e., with longer deadlines of the aperiodic jobs.

[10]The runtime of the partitioned algorithm serves as reference.

Chapter 6

Effectiveness Evaluation

In this chapter, we aim to quantify the effectiveness of the slot shifting algorithm. All experiments described in this chapter are performed on 4 or on 32 cores since the global algorithms show their benefits only in multicore environments. In order to run many different experiments with thousands of job sets, a much faster implementation is required. For that reason, we employ a multi-threaded Linux-based implementation instead of the previously used MPARM-based implementation.

In a first step, we create many thousands of random job sets for a multitude of settings. Then, we construct and annotate the corresponding offline scheduling tables, as described in section 2.2. After that, our Linux-based implementation runs the online phase of the slot shifting algorithms based on these tables. Finally, we collect the results to evaluate the effectiveness of the algorithms. Therefore, we determine 95% confidence intervals for the maximum achievable acceptance ratio, analyze the average number of performed acceptance tests, and measure the resulting response times of the jobs of the aperiodic tasks.

The remainder of this chapter is structured as follows: First, we describe the implementation of the experiments and we list the variety of parameters that influence the experiments. After that, we describe the individual experiments and their results. At the end of the section, we summarize the findings of our experiments.

6.1 Implementation

Our experiments are based on the same code as we used before in section 5 to evaluate the efficiency of slot shifting on the MPARM simulator. For speed reasons, this code does not *schedule* any real workload jobs. We perform the experiments described in this section on Elwetritsch, the high performance cluster of the University of Kaiserslautern. Elwetritsch offers a large pool of multicore machines running Scientific Linux release 6.3 (Carbon) with Kernel version 2.6.32 for 64-bit architectures.

For these experiments, our implementation includes the partitioned slot shifting algorithm and the two global slot shifting algorithms; all three in a "normal" version and in a version with SDL. Furthermore, this implementation includes a table driven scheduler. If multiple offline jobs are ready to execute, this scheduler makes its decision based on the EDF algorithm. Furthermore, incoming aperiodic jobs are scheduled when

the processor idles, i.e., this implementation of EDF employs background processing of aperiodic jobs. To summarize, we run the experiments with the following scheduling algorithms:

- EDF based offline scheduler with background processing of aperiodic jobs
- partitioned slot shifting (normal/with SDL)
- global algorithm 1: spare-capacity-based slot shifting (normal/with SDL)
- global algorithm 2: negotiation-based slot shifting (normal/with SDL)

If not mentioned otherwise, we vary the offline utilizations per core between 0% and 90% in steps of 10% in each distinct experiment. The offline utilizations per core is equally balanced on all cores of the system. An important parameter that influences the results is the utilization that is created per core by the incoming aperiodic jobs. As before in section 5.2, we define the utilization of the aperiodic jobs, $U_{aperiodic}$, as the sum of their worst case execution times divided by the length of the offline scheduling table. That is, $U_{aperiodic}$ is the fraction of time that the core will potentially spend to process the aperiodic jobs throughout the length of the scheduling table. We repeat all experiments three times to obtain results for the following values of $U_{aperiodic}$: 10%, 20%, and 50%. All aperiodic jobs arrive randomly throughout the time interval given by the length of the offline table. Except for Experiment 5 which analyzes different arrival patterns, the utilization created by the aperiodic jobs is equally balanced on all cores of the system.

Another important parameter is the relation of the deadline of the aperiodic jobs to their WCETs, which is given by the DLX factor. While all other parameters remain unchanged, we repeat Experiments 1, 2, 3, and 5 with DLX factors of 1, 1.5, 2, and 5. Experiment 4 uses the following DLX factors: 1.5, 2, 5, 10, 15, 20. Note that we modify the deadlines of the aperiodic jobs of jobs sets with DLX factor of 1 slightly: Their deadlines are equal to their worst case execution times plus one slot, thus a single delegation to another core is possible if local acceptance fails.

Each distinct result for each applied algorithm and for one specific set of parameters (offline utilization, aperiodic utilization, DLX factor) is based on 1000 randomly generated job sets. Hence, for Experiments 1, 2, 3, and 5 in total 120000 job sets have been run with the seven distinct algorithms listed above. For Experiment 4, in total 240000 job sets have been run with these seven algorithms.

To facilitate comparability of the results, the same job sets are used as input for all algorithms listed above with a fixed parameter setting (offline utilization, aperiodic utilization, DLX factor).

Performing an experiment consists of 5 steps:

In the first step, the tool described in section 5.2.1.1 creates thousands of random job sets based on the desired parameters.

In the second step, we employ the offline phase of slot shifting as described in chapter 5.2.1.2. Thus, we obtain for each individual job set an offline scheduling table which maps the jobs to cores and resolves the jobs precedence and deadline constraints. Furthermore, the offline phase annotates the offline scheduling table. After this step, the

table additionally contains interval and spare capacity information. Note that we create the jobs sets in the first step such that the resulting offline scheduling tables from the second step feature lengths between 500 and 5000 slots.

In the third step, we compile the source code to obtain the executable for the online phase of slot shifting.

In the fourth step, the actual experiment is performed by running the online phase of slot shifting for a single table length. Our multi-threaded implementation of the online phase runs one of the different aforementioned seven scheduling algorithms.

In the fifth and last step, we collect the intermediate results from the thousands of job sets and distill the final result files. This includes the calculation of, e.g., the values of the mean acceptance ratio and the values of the upper and lower bounds of the 95% confidence intervals.

Due to the vast amount of results for each individual experiment, we will exemplify our findings based on a sub set of the results. The interested reader may refer to technical report [81] or to Appendix C which list all results of all measurements for each experiment in great detail.

6.2 Experiments on Linux

6.2.1 Experiment 1: Balanced Utilization, 4 Cores

In Experiment 1, we evaluate the effectiveness of the different versions of the slot shifting algorithm on 4 cores. As already mentioned, we run the following four algorithms in a normal version, and with SDL, if applicable: EDF with background processing of aperiodic tasks, partitioned slot shifting, spare-capacity-based multicore slot shifting (global alg. 1), and negotiation-based multicore slot shifting (global alg. 2). The parameters of the task sets can be found in Table 6.1. The utilization created by the jobs of the offline tasks, $U_{offline}$, is the same, i.e., balanced, on all cores and varied between 0% and 90% in steps of 10%[1]. We run the experiment three times with an aperiodic utilization per core of 10%, 20%, or 50%, respectively. These parameters allow us to evaluate the capabilities of the different algorithms to integrate aperiodic jobs under low as well as under high system utilization. We use the parameter settings described in Table 6.1 also as the basis the following experiments and mention the modifications that apply.

We only present the results for DLX factor 2, Appendix C also lists the detailed results for the DLX factors 1, 1.5, and 5, see Tables C.2–C.4.

Parameter	Offline Tasks	Aperiodic Tasks
worst case execution time	$1-15$	$10-15$
period	$15-30$	–
deadline	$15-30$	$2*C$
resulting U per core	0%, 10%, 20%, ... 90%	10%, 20%, 50%
max. allowed deviation from target U	<1%	

Table 6.1: *Experiment 1: overview of the task set parameters, DLX factor 2.*

Figure 6.1, 6.2, and 6.3 show the resulting acceptance ratios of the different algorithms for the case of 10%, 20%, and 50% aperiodic job utilization, respectively. Each value of the acceptance ratio is based on 1000 job sets. In the case of 10% aperiodic job utilization, as long as the utilization created by the jobs of the offline tasks is low (0% and 10%), all algorithms perform similarly well.
As $U_{offline}$ increases (20% – 40%), the acceptance ratio of partitioned slot shifting and even more that of EDF with background processing decrease, while both global algorithms manage to keep a high acceptance ratio. When the utilization increases further, the acceptance ratio of the EDF-based scheduler rapidly drops towards zero, while the slot shifting algorithms manage to keep higher acceptance ratios: Both global algorithms show approximately the same acceptance ratio and outperform the partitioned algorithm.
With $U_{offline}$ reaching 90%, the acceptance ratio of the EDF-based scheduler is 0.04%,

[1]The job sets have been created as described previously in section 5.2.1.

that of partitioned slot shifting is 10.7%, and that of both global algorithms is still approximately 23%.

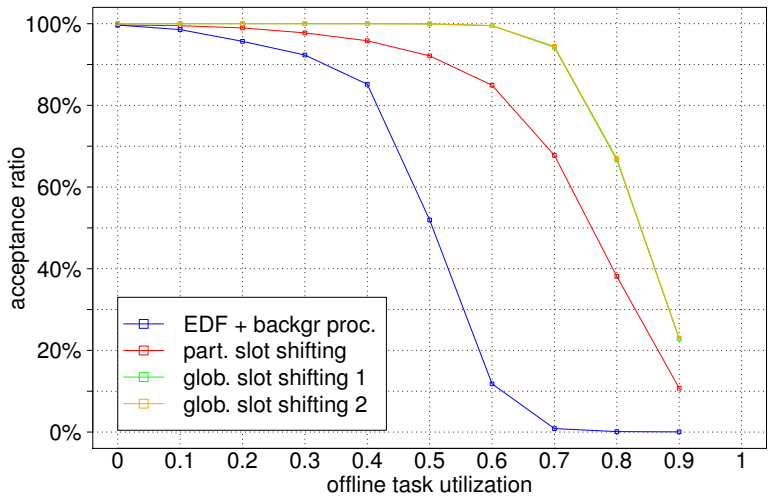


Figure 6.1: *Experiment 1: acceptance ratio, $U_{aperiodic} = 10\%$, DLX factor 2.*

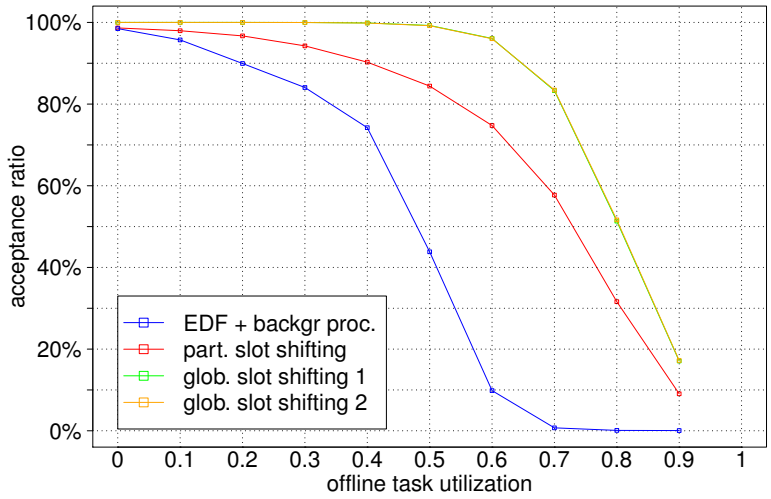


Figure 6.2: *Experiment 1: acceptance ratio, $U_{aperiodic} = 20\%$, DLX factor 2.*

Similar observations can be made in the cases of 20% and 50% aperiodic job utilization. A general trend is visible: With increasing "pressure" created by the aperiodic jobs (from $U_{aperiodic} = 10\%$, over 20% to 50%), the acceptance ratio of all algorithms generally drops quicker and slightly deeper. Interestingly, in the case of $U_{aperiodic} = 50\%$, partitioned

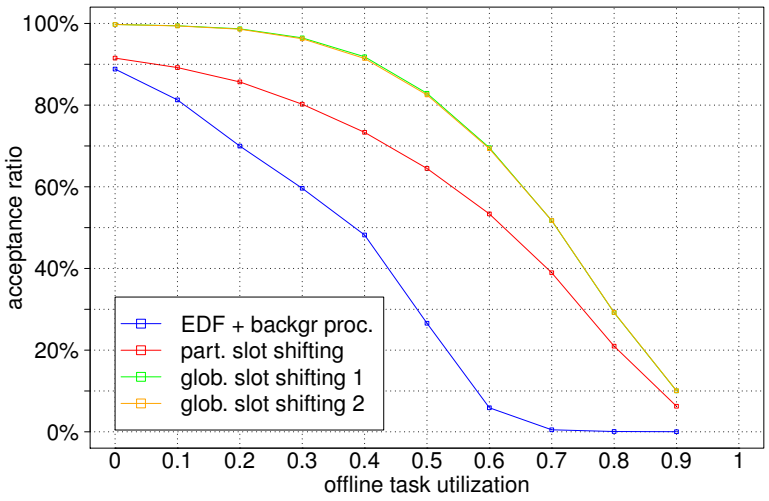


Figure 6.3: *Experiment 1: acceptance ratio, $U_{aperiodic}$ = 50%, DLX factor 2.*

slot shifting fails to integrate all aperiodic jobs even in the absence of offline jobs. The reason is that when multiple aperiodic jobs arrive on the same core shortly after each other, some of them cannot be accepted. Under the same circumstances, the global algorithms delegate these jobs to other cores and thus successfully integrate them.

In general, the experiment shows that the partitioned slot shifting algorithm always performs better (i.e., features a higher acceptance ratio) than EDF with background processing of aperiodic jobs. Furthermore, it shows that both global algorithms outperform their partitioned counterpart. The acceptance ratio of the spare-capacity-based global slot shifting algorithm is always slightly smaller than that of the negotiation-based global slot shifting algorithm.

We also calculated the limits of the 95% confidence intervals for each of the mean acceptance ratios of this experiment. These limits cannot be visualized in Figure 6.1, 6.2, and 6.3 as they are too tight to be plotted. Therefore, Table 6.2 brings their exact values for the case of 20% aperiodic job utilization and a DLX factor of 2; for the other cases, the values are similar (see Appendix C, Tables C.5– C.7). As can be seen from Table 6.2, almost all limits are much below 0.5%, i.e., creating 95% confidence intervals with lengths much smaller than 1%. The biggest value for the limits of the confidence interval occurs at 80% offline utilization for global slot shifting algorithm 1 with 0.699%. This results in a 95% confidence interval for the mean acceptance ratio of [50.646, 52.044], i.e., of an approximate length of only 1.4%. Figure 6.4 highlights the improvement of the acceptance ratio of the global algorithms compared to the partitioned slot shifting for 10%, 20%, and 50% aperiodic job utilization. In general, both global algorithms show very similar improvements compared to the partitioned algorithm; at 10% and 20% aperiodic utilization, global algorithm 2 is slightly better, at 50% global algorithm 1 outperforms its counterpart. Independent of the utilization values created by offline or aperiodic jobs, using global algorithms improves the acceptance ratio.

Offline Utilization	EDF with Backgr. Proc.	Partitioned Slot Shifting	Global Slot Shifting 1	Global Slot Shifting 2
0%	98.484 ± 0.073	98.645 ± 0.062	99.999 ± 0.001	100.000*
10%	95.714 ± 0.118	97.965 ± 0.078	99.999 ± 0.001	99.999 ± 0.001
20%	89.942 ± 0.176	96.698 ± 0.098	99.998 ± 0.002	99.998 ± 0.002
30%	84.058 ± 0.209	94.256 ± 0.127	99.980 ± 0.007	99.975 ± 0.008
40%	74.224 ± 0.238	90.267 ± 0.157	99.871 ± 0.022	99.853 ± 0.025
50%	43.827 ± 0.265	84.412 ± 0.176	99.250 ± 0.053	99.199 ± 0.054
60%	9.839 ± 0.171	74.754 ± 0.254	96.076 ± 0.131	96.044 ± 0.129
70%	0.718 ± 0.058	57.724 ± 0.463	83.279 ± 0.419	83.494 ± 0.405
80%	0.092 ± 0.017	31.616 ± 0.561	51.345 ± 0.699	51.578 ± 0.697
90%	0.054 ± 0.015	9.057 ± 0.298	17.095 ± 0.422	17.213 ± 0.425

Table 6.2: *Experiment 1: mean acceptance ratios (in %) with 95% confidence intervals (20% aperiodic utilization, DLX factor 2). (*) Note that at $U_{offline}$ = 0% for global slot shifting 2, all aperiodic jobs in all job sets have been accepted.*

Nevertheless, $U_{aperiodic}$ influences this improvement: While at 10% aperiodic job utilization, the maximum improvement results at a higher offline utilization value (80%), with increasing aperiodic utilization, the maximum improvement results at lower offline utilizations: 70% and 40% for 20% and 50% aperiodic job utilization.

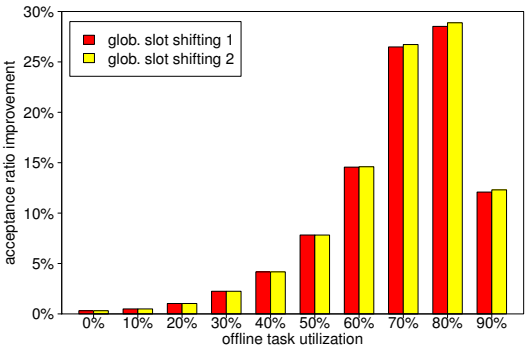
Another interesting metric to evaluate the performance of the algorithms is the mean value of the number of acceptance tests performed per aperiodic job. Table 6.3 lists these values for different utilization values separated into the following categories: accepted aperiodic jobs, finally rejected aperiodic jobs, and total mean of all aperiodic jobs. For both global algorithms, the table lists these values for jobs sets with DLX factor 2 and an aperiodic job utilization of 20%; Appendix C lists the resulting values for the other DLX factors and values of $U_{aperiodic}$ in Tables C.8–C.16. The data for each entry is again based on 1000 job sets.

Notice that for the categories *rejected* and *accepted* this table makes no statement

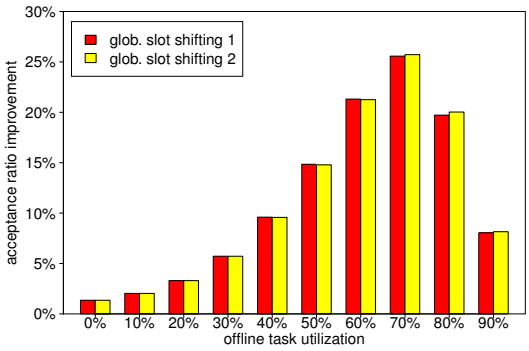
Type	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
accepted	global 1	1.01	1.02	1.03	1.06	1.11	1.19	1.32	1.50	1.66	1.73
	global 2	1.02	1.03	1.04	1.09	1.15	1.28	1.50	1.84	2.17	2.41
rejected	global 1	4.00	4.00	4.00	4.00	3.99	3.99	3.99	3.98	3.97	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
total	global 1	1.01	1.02	1.03	1.06	1.11	1.21	1.42	1.91	2.78	3.57
	global 2	1.02	1.03	1.04	1.09	1.16	1.30	1.60	2.20	3.05	3.72

Table 6.3: *Experiment 1: mean value of the number of acceptance tests per aperiodic job for $U_{aperiodic}$ = 20%, DLX factor 2.*

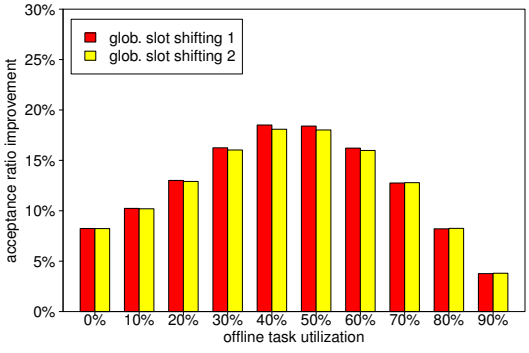
about how many of the 1000 job sets have been considered, i.e., how often, e.g., a final rejection of a job occurs.
For example at 0% offline utilization, the mean value of 4.00 tests per finally rejected aperiodic job for global algorithm 2 is only based on 1 of 1000 job sets. The other 999 job sets could successfully integrate all aperiodic jobs, hence they do not contribute to this table entry. Furthermore, in this single job set that has been counted, the vast majority of all aperiodic jobs (i.e., 212 of 213) have been accepted, and only a single aperiodic job has been finally rejected. Thus, these values presented in this table have to be read with the previously presented acceptance ratios in mind. In general, the same conclusions can be drawn when the DLX factor is varied. The more the DLX factor increases, the less tests are performed to integrate the aperiodic jobs. Increasing $U_{aperiodic}$ has the opposite effect, since more aperiodic jobs make it harder for slot shifting to integrate them successfully.



(a) *10% aperiodic utilization.*



(b) *20% aperiodic utilization.*



(c) *50% aperiodic utilization.*

Figure 6.4: *Experiment 1: improvement of mean acceptance ratio: global vs. partitioned algorithms on 4 cores, DLX factor 2.*

We also measure the response times of the aperiodic jobs. To compare the results, we normalize the response times of all aperiodic jobs in all job sets. For that purpose, we define the so called *quickness*, a value in the range of [0,1], as:

$$Q = 1 - \frac{R - C}{D - C} \tag{6.1}$$

with R the measured response time of the aperiodic job, C its worst case execution time and D its deadline. Q becomes 1 if the response time of the aperiodic job is equal to its worst case execution time C, and Q becomes 0 if the response time of the aperiodic job is equal to its deadline D. In other words, results closer to 1 indicate better responsiveness of the aperiodic jobs.

Table 6.4 lists the mean value of the quickness for all three aperiodic utilization values (10%, 20%, and 50% aperiodic utilization) with a DLX factor of 2. Values for other DLX factors are listed in Appendix C, in Table C.17, C.18, and C.19. Independent of the utilization created by the aperiodic jobs, there is a general trend visible: Their mean response time is best, when no offline jobs are present. With increasing number of offline jobs, the quickness decreases, i.e., the response time increases. This trend is however not perfectly linear: The local optimum of the response time, e.g., in case of $U_{aperiodic} = 20\%$, occurs for EDF at an offline utilization of 60%, and for the partitioned as well as for both global slot shifting algorithms at 80%.

$U_{aperiodic}$	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
10%	EDF	0.96	0.85	0.73	0.58	0.40	0.26	0.18	0.33	0.27	0.22
	partitioned	0.96	0.93	0.89	0.83	0.77	0.69	0.60	0.53	0.51	0.55
	global 1	0.96	0.93	0.89	0.83	0.77	0.69	0.59	0.52	0.47	0.51
	global 2	0.96	0.92	0.89	0.83	0.77	0.68	0.58	0.51	0.47	0.50
20%	EDF	0.91	0.81	0.70	0.56	0.39	0.26	0.19	0.32	0.27	0.20
	partitioned	0.91	0.88	0.84	0.79	0.73	0.66	0.57	0.51	0.49	0.54
	global 1	0.91	0.88	0.84	0.78	0.72	0.65	0.55	0.47	0.44	0.48
	global 2	0.91	0.87	0.83	0.78	0.71	0.64	0.54	0.46	0.43	0.48
50%	EDF	0.79	0.70	0.62	0.51	0.37	0.26	0.20	0.32	0.22	0.17
	partitioned	0.79	0.75	0.71	0.67	0.63	0.58	0.51	0.46	0.46	0.51
	global 1	0.77	0.72	0.67	0.61	0.54	0.47	0.40	0.37	0.38	0.43
	global 2	0.75	0.71	0.65	0.59	0.52	0.46	0.40	0.36	0.38	0.44

Table 6.4: *Experiment 1: mean value of the quickness, DLX factor 2.*

Figures 6.5–6.12 show the *distribution* of the quickness of the aperiodic jobs for different offline utilizations between 0% and 90% in steps of 10%. These histograms are obtained for an aperiodic job utilization of 20% and a DLX factor of 2; similar histograms for aperiodic job utilization values of 10% and 50% can be found in Appendix C, see Figures C.1–C.16.

Figure 6.5 and Figure 6.6 show the distribution of the quickness of the aperiodic jobs for EDF with background processing. Without the presence of any offline jobs,

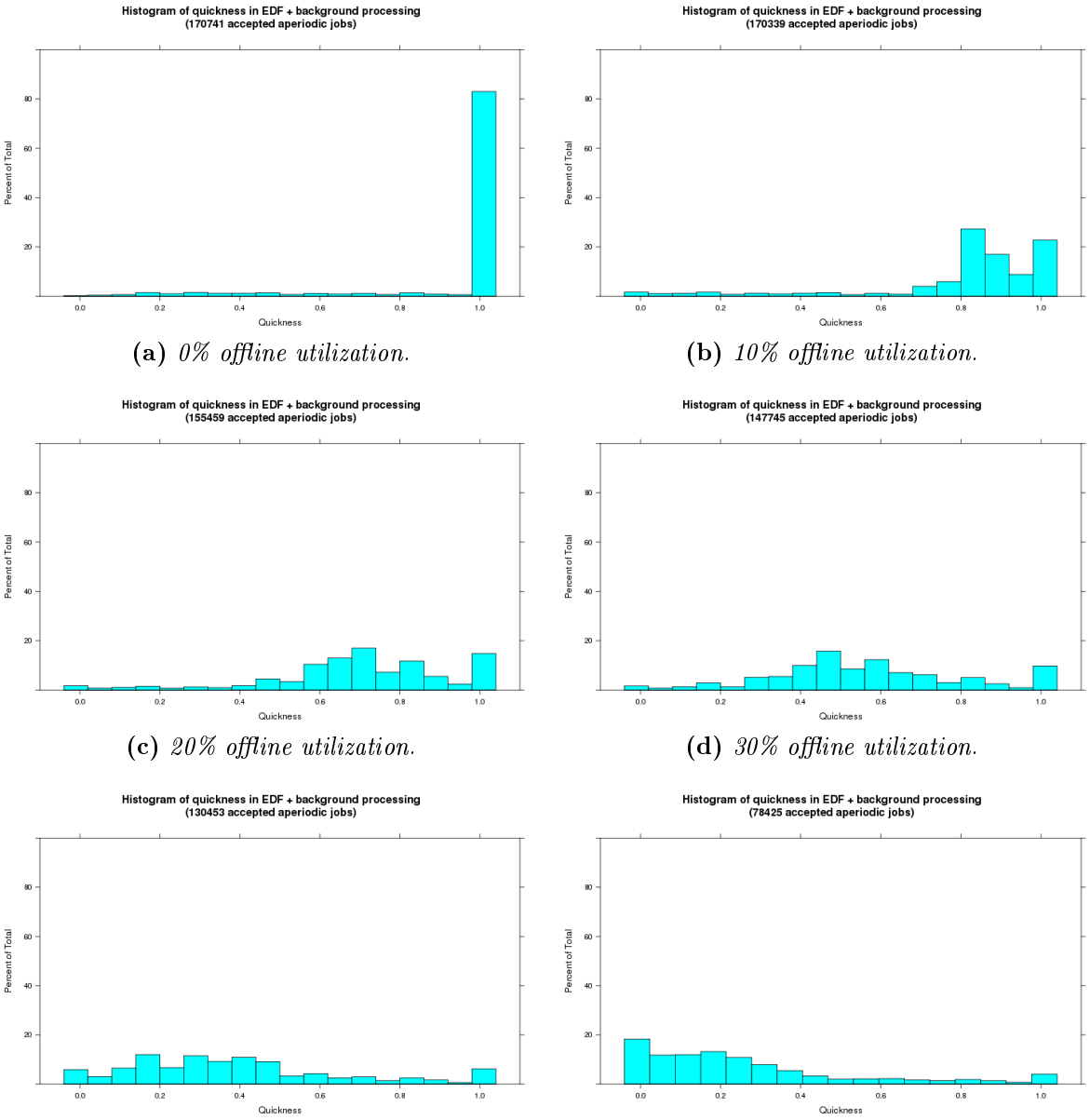
most aperiodic jobs feature a very short response time, i.e., a quickness close to 1 (see Figure 6.5a). With increasing offline utilization, the response times of the aperiodic jobs grow and become much more equalized (see Figure 6.5b to Figure 6.5e). As the offline utilization keeps increasing, more and more aperiodic jobs feature lower quickness values. At 60% offline utilization, many jobs feature a quickness close to 0, i.e., a long response time (see Figure 6.6a). When the offline utilization reaches 90%, approximately half of the aperiodic jobs feature a minimum quickness value, i.e., finish execution exactly at their deadline (see Figure 6.6d). The reason for the decreased number of bins in the last histogram plot is that only very few aperiodic jobs—76 jobs in total—have been scheduled successfully[2].

Figure 6.7 and 6.8 show the distribution of the quickness of the aperiodic jobs for partitioned slot shifting. The aperiodic jobs experience a similar effect as seen before with EDF with background processing: With very low offline utilization (Figure 6.7a), most jobs feature a very quick response time, noticeable as peak on the right side of the histogram. This peak slowly shrinks as the offline utilization increases to 90%.

Figure 6.9 and 6.10 show the distribution of the quickness of the aperiodic jobs for the global slot shifting algorithm 1. The figures are very similar to those of the partitioned slot shifting algorithm, shown in Figure 6.7 and 6.8, respectively.
The most important difference between the histograms for the partitioned slot shifting algorithm and for the global slot shifting algorithm 1 is that the partitioned algorithm seems to slightly perform better than the global algorithm, since more jobs feature an optimum response time. At 60% utilization, the partitioned algorithm integrates 20.5% of the aperiodic jobs (27896 jobs) with an optimum (i.e., minimum) response time. At the same offline utilization, global algorithm 1 integrates 12.9% of the aperiodic jobs (22703 in absolute numbers) such that their response time is optimum.
At 90% utilization, the partitioned algorithm integrates 15.8% of the aperiodic jobs (2710 jobs) with an optimum (minimum) response time, while the global algorithm algorithm integrates 4.2% of the aperiodic jobs (1352 in absolute numbers) such that their response time is optimum. Nevertheless, when considering the total numbers, the global algorithms always integrate more aperiodic jobs than the partitioned slot shifting algorithm (e.g., for 90% offline utilization: global: 32218 jobs, partitioned: 17121 jobs).

Figure 6.11 and 6.12 show the distribution of the quickness of the aperiodic jobs for global slot shifting algorithm 2. They are nearly identical to these of global slot shifting algorithm 1 that we explained before.

[2]Note that we use R's standard setting, more specifically, Sturges' formula [82], to define the number of bins for the histogram plot.



(a) *0% offline utilization.*

(b) *10% offline utilization.*

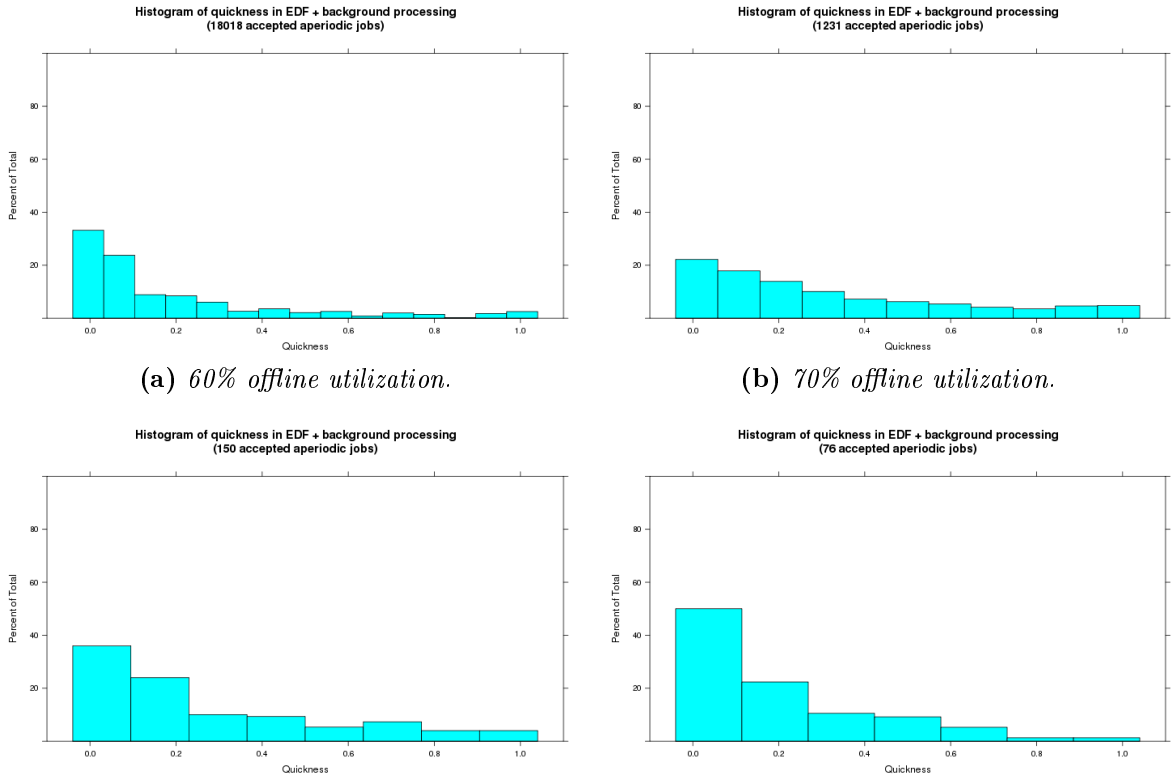
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

(f) *50% offline utilization.*

Figure 6.5: *Experiment 1: histograms of quickness of EDF with background processing of aperiodic jobs, $U_{aperiodic} = 20\%$, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



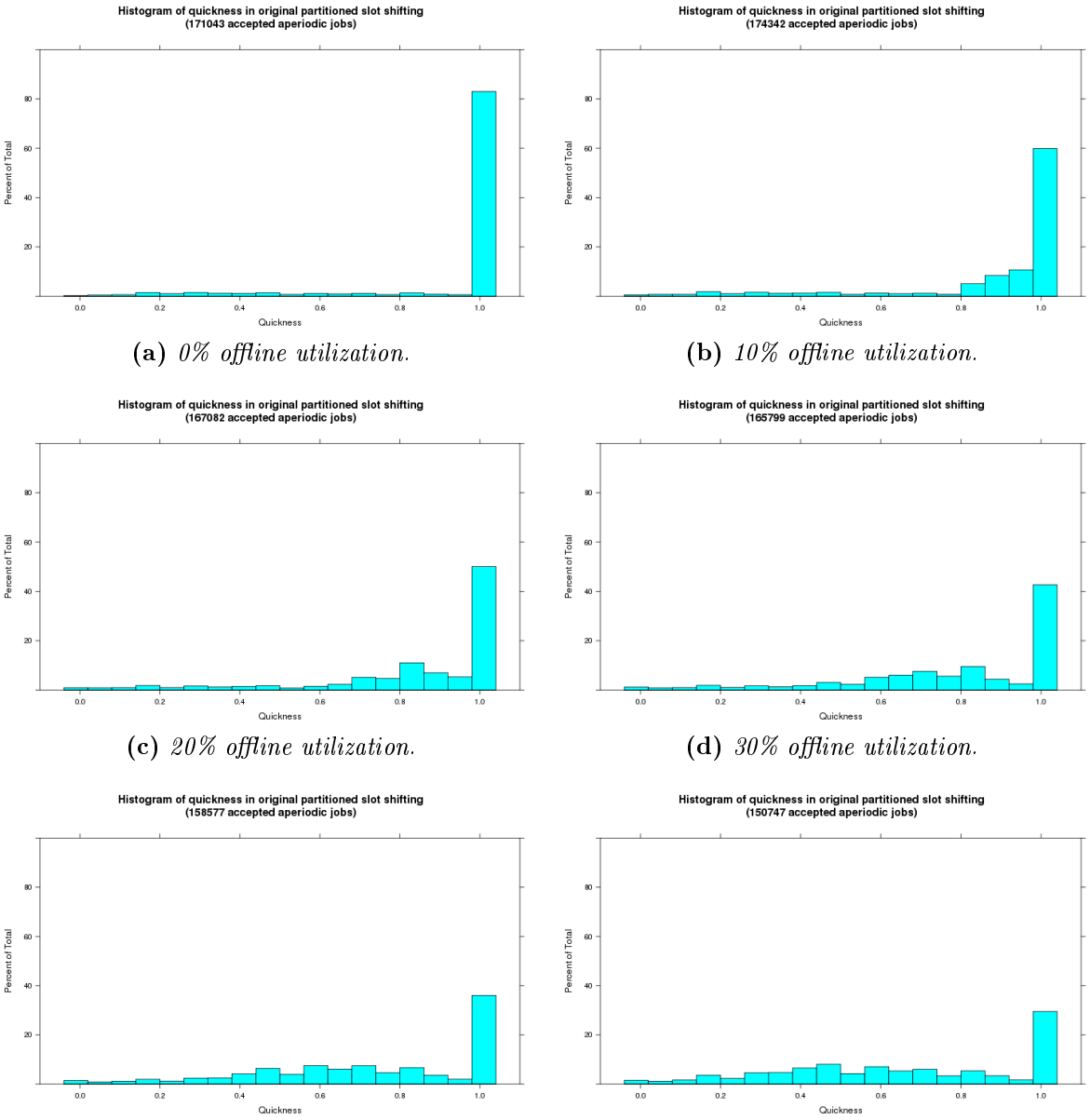
(a) *60% offline utilization.*

(b) *70% offline utilization.*

(c) *80% offline utilization.*

(d) *90% offline utilization.*

Figure 6.6: *Experiment 1: histograms of quickness of EDF with background processing of aperiodic jobs, $U_{aperiodic} = 20\%$, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *0% offline utilization.*

(b) *10% offline utilization.*

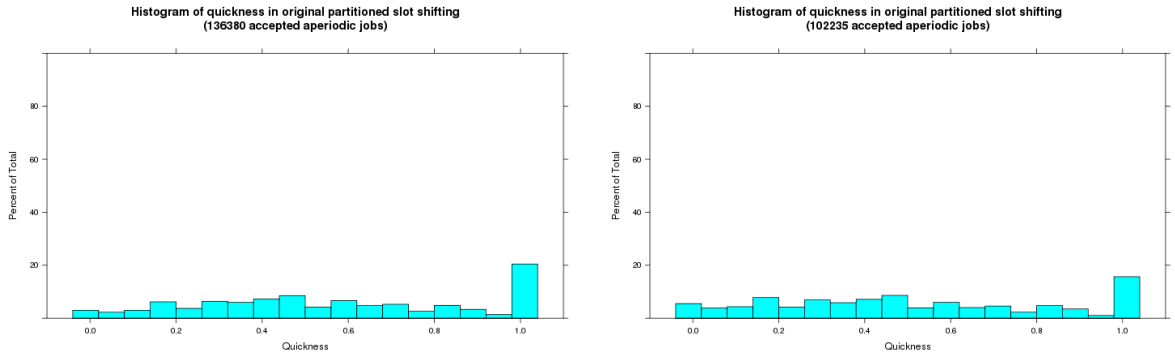
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

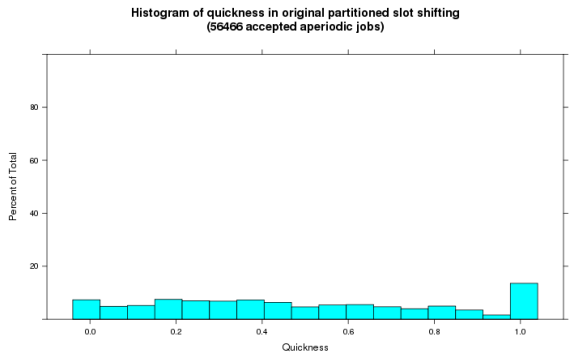
(f) *50% offline utilization.*

Figure 6.7: *Experiment 1: histograms of quickness of the partitioned slot shifting algorithm, $U_{aperiodic}$ = 20%, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*

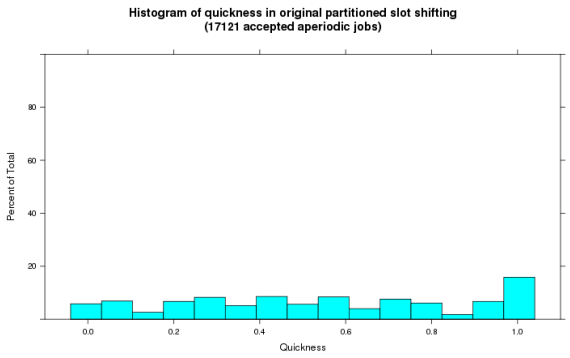


(a) *60% offline utilization.*

(b) *70% offline utilization.*

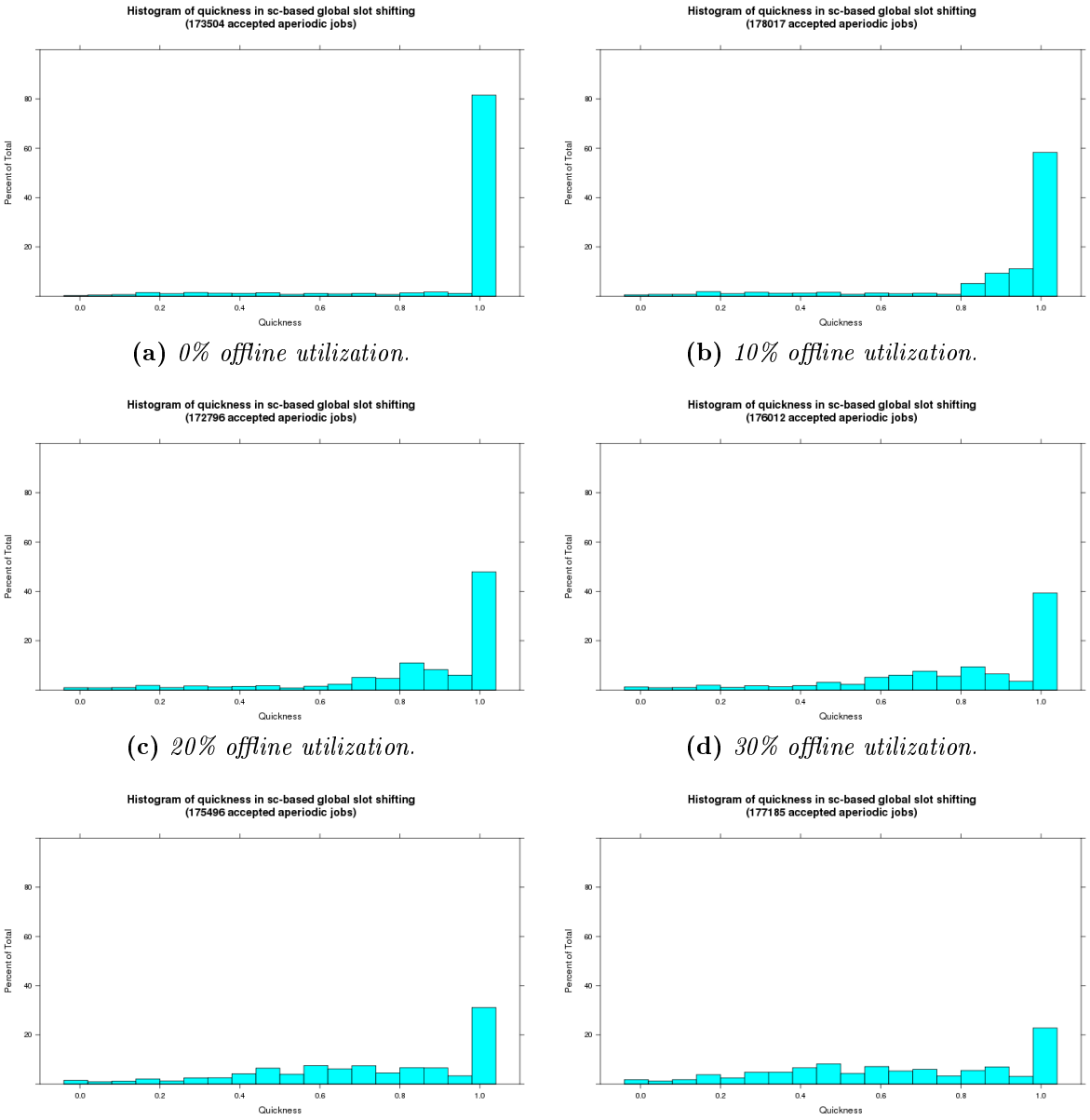


(c) *80% offline utilization.*



(d) *90% offline utilization.*

Figure 6.8: *Experiment 1: histograms of quickness of the partitioned slot shifting algorithm, $U_{aperiodic} = 20\%$, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *0% offline utilization.*

(b) *10% offline utilization.*

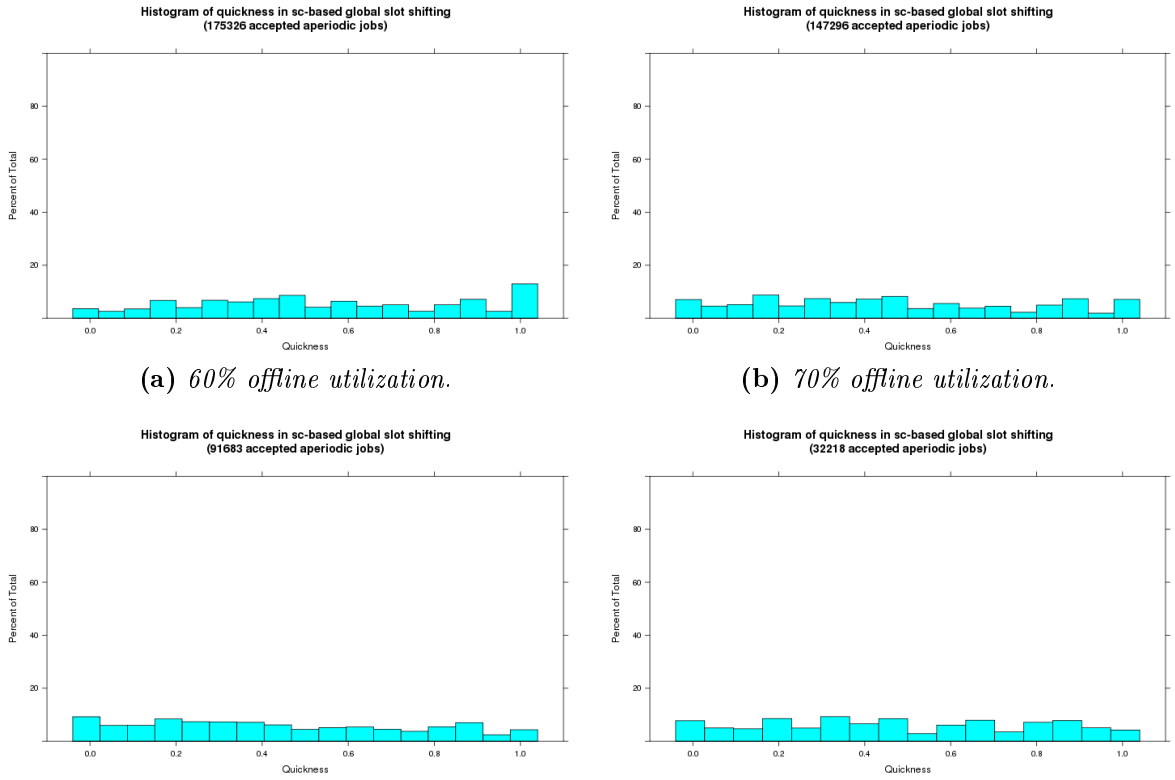
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

(f) *50% offline utilization.*

Figure 6.9: *Experiment 1: histograms of quickness of the spare-capacity-based slot shifting algorithm, $U_{aperiodic}$ = 20%, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



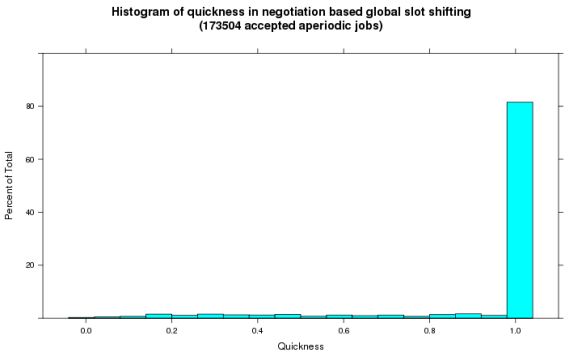
(a) *60% offline utilization.*

(b) *70% offline utilization.*

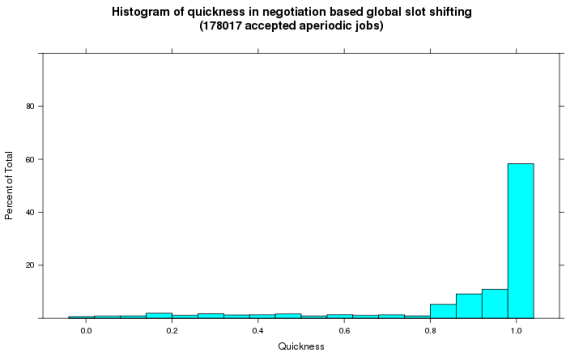
(c) *80% offline utilization.*

(d) *90% offline utilization.*

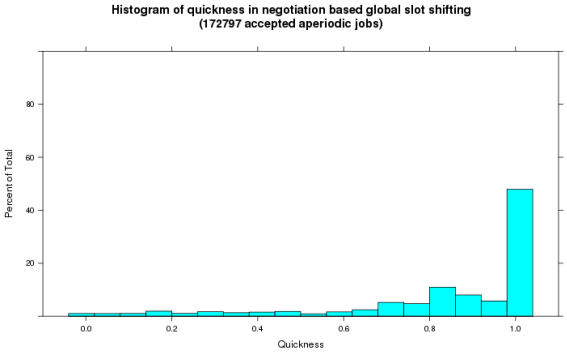
Figure 6.10: *Experiment 1: histograms of quickness of the spare-capacity-based slot shifting algorithm, $U_{aperiodic} = 20\%$, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



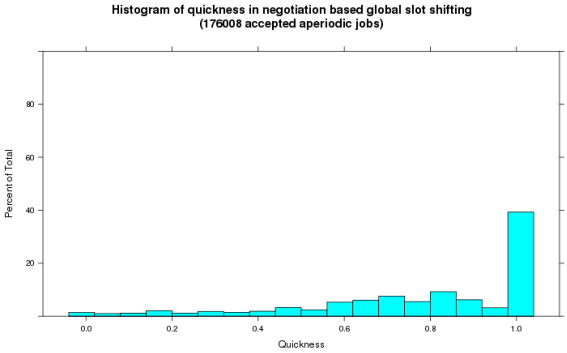
(a) *0% offline utilization.*



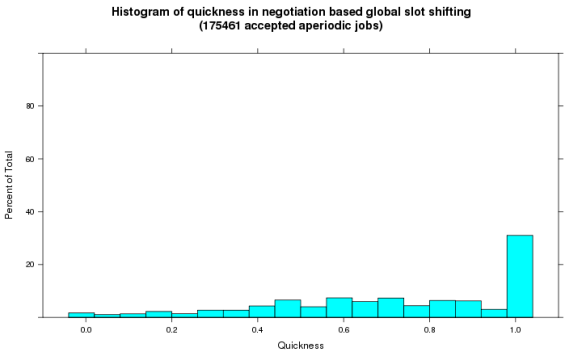
(b) *10% offline utilization.*



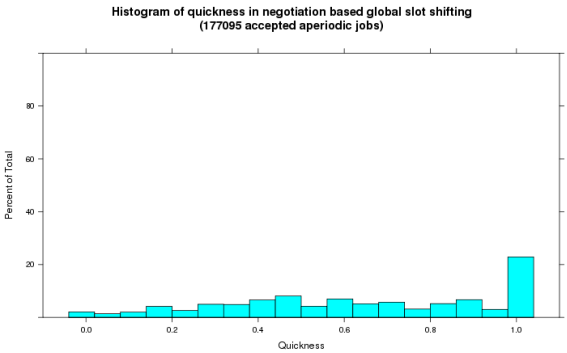
(c) *20% offline utilization.*



(d) *30% offline utilization.*

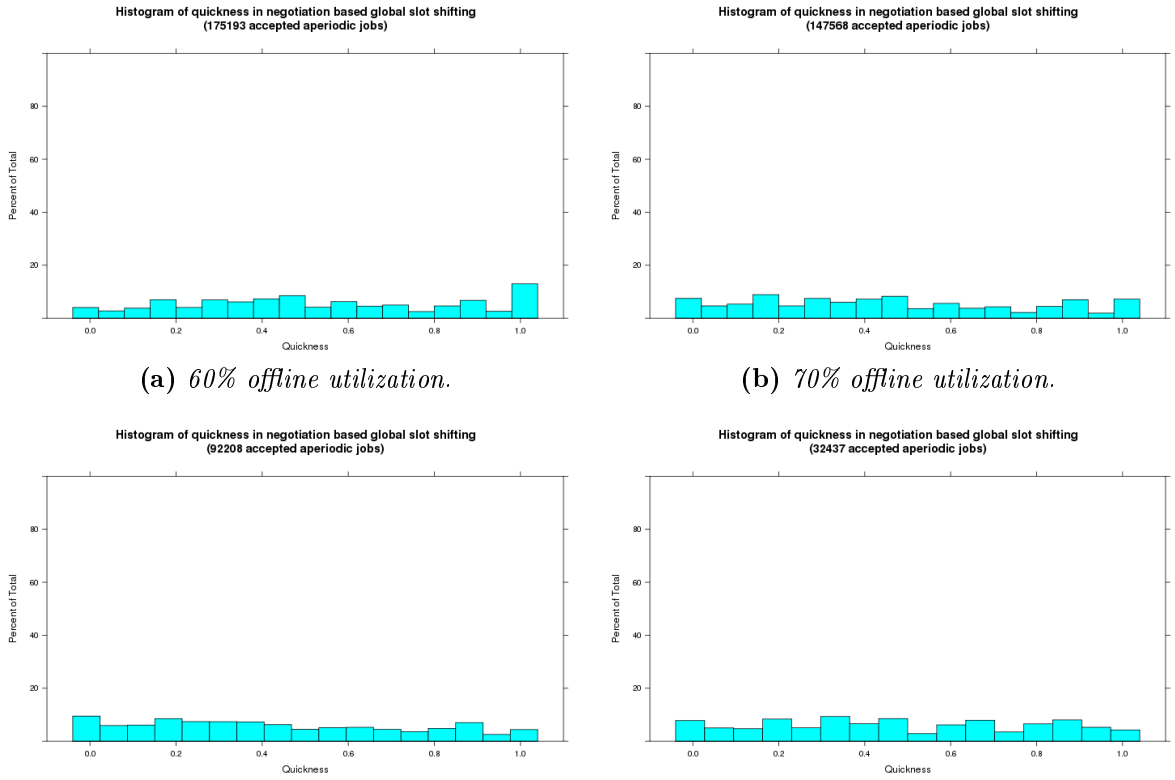


(e) *40% offline utilization.*



(f) *50% offline utilization.*

Figure 6.11: *Experiment 1: histograms of quickness of the negotiation-based slot shifting algorithm, $U_{aperiodic}$ = 20%, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *60% offline utilization.*

(b) *70% offline utilization.*

(c) *80% offline utilization.*

(d) *90% offline utilization.*

Figure 6.12: *Experiment 1: histograms of quickness of the negotiation-based slot shifting algorithm, $U_{aperiodic}$ = 20%, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*

6.2.2 Experiment 2: Balanced Utilization, 32 Cores

In this experiment, we repeat Experiment 1 on 32 cores. The exact values of the mean acceptance ratio are listed in Tables D.1–D.3 in Appendix D. Figure 6.13, 6.14, and 6.15 show the resulting acceptance ratios of the different algorithms for the case of 10%, 20%, and 50% aperiodic job utilization, respectively. The resulting curves look similar to these shown in Figures 6.1, 6.2, and 6.3. As expected, the results for EDF with background processing of aperiodic jobs and for partitioned slot shifting are the same as in Experiment 1, i.e., they are independent of the number of cores. All three figures show: On a system consisting of 32 cores, the global slot shifting algorithms generally achieve better acceptance ratios than in Experiment 1.

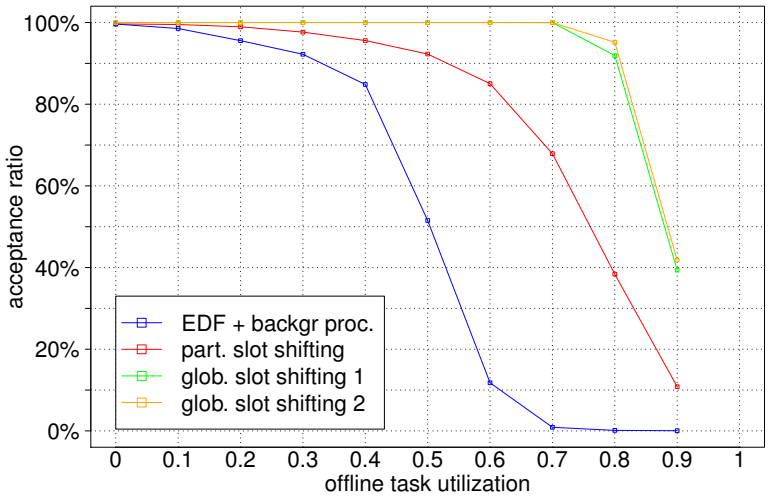


Figure 6.13: *Experiment 2: acceptance ratio, $U_{aperiodic}$ = 10%, DLX factor 2.*

As in the previous experiment, we calculate the limits of the 95% confidence intervals for the mean acceptance ratio. And as before, these limits cannot be visualized in Figure 6.13, 6.14, and 6.15 as they are too tight to be plotted. Their exact values are listed in Appendix D, Table D.4, D.5, and D.6.

Figure 6.16 shows the improvement of the acceptance ratio from 4 to 32 cores for the three distinct scenarios. As long as the utilization created by offline jobs is low, there is no need to delegate aperiodic jobs to other cores. The figures show zero or very low improvements in this case, since there is no benefit in having more cores available.
With increasing utilization created by offline jobs, more and more aperiodic jobs are delegated as local acceptance fails. The figures show up to 30% improvement of the acceptance ratio compared to Experiment 1.

Table 6.5 lists the mean number of acceptance tests performed per aperiodic job in the following categories: accepted aperiodic jobs, finally rejected aperiodic jobs, and total

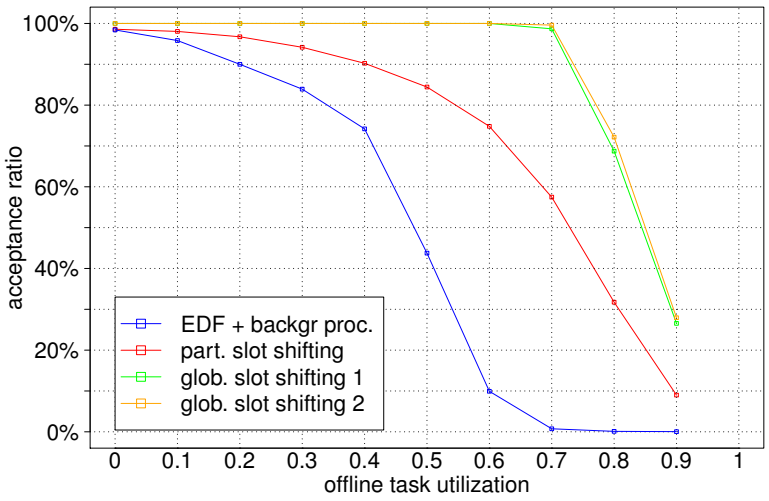


Figure 6.14: *Experiment 2: acceptance ratio, $U_{aperiodic}$ = 20%, DLX factor 2.*

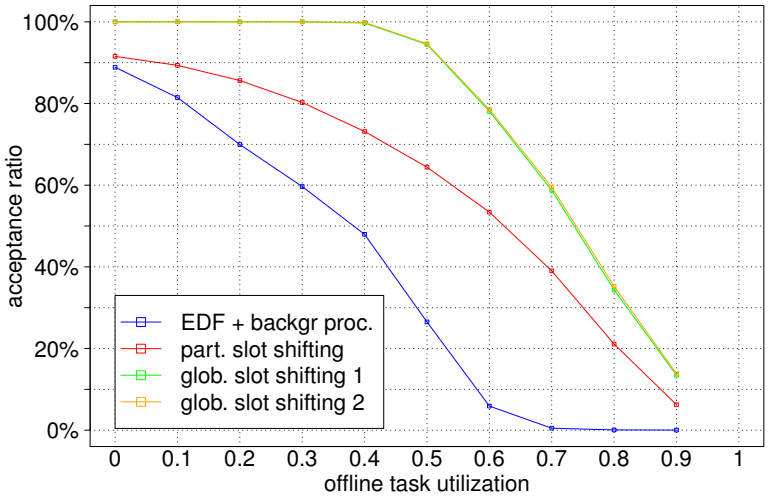
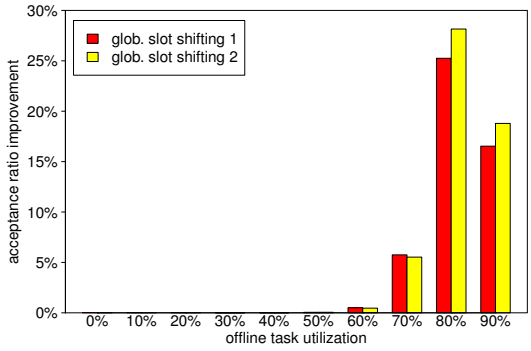


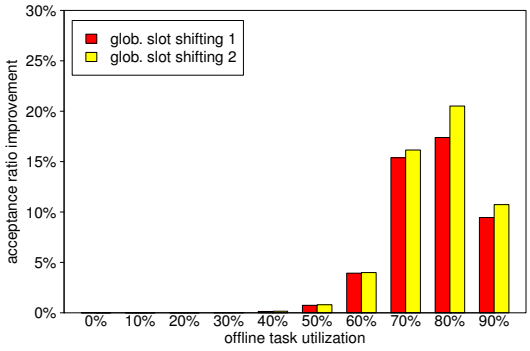
Figure 6.15: *Experiment 2: acceptance ratio, $U_{aperiodic}$ = 50%, DLX factor 2.*

mean value of all aperiodic jobs. The table lists the results for $U_{aperiodic}$ = 20% and a DLX factor of 2; Tables D.7–D.15 in Appendix D list more results. We omit the results for partitioned slot shifting as they are identical to those of the previous experiment.

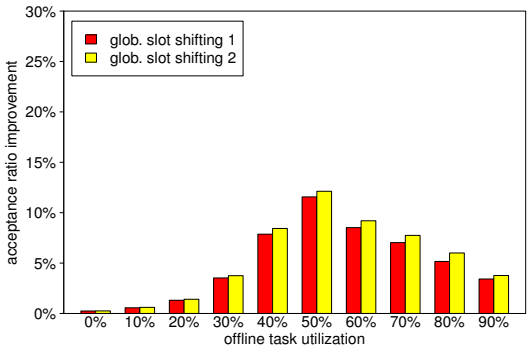
In the first category, the average number of performed acceptance tests is approximately 1 for low offline utilizations. The value rapidly increases when the offline utilization crosses the 60% margin; at 90% the value reaches 2.74 for global algorithm 1 and 13.67 for global algorithm 2. On the one hand, this shows that the former algorithm is able to accept approximately the same number of aperiodic jobs performing much less acceptance tests. On the other hand, this shows that the latter algorithm—even under high system utilization—on average does not perform as many acceptance tests for the



(a) $U_{aperiodic} = 10\%$.



(b) $U_{aperiodic} = 20\%$.



(c) $U_{aperiodic} = 50\%$.

Figure 6.16: *Experiment 2: acceptance ratio improvement, 4/32 cores, DLX factor 2.*

aperiodic jobs as there are cores in the system.
In the second category, the table shows that for offline utilizations below 60% (70% for algorithm 2) both algorithms manage to accept all aperiodic jobs. When the offline utilization rises further, global algorithm 2, as expected, performs for finally rejected aperiodic jobs as many acceptance tests as there are cores in the system. When the utilization becomes larger than 60%, global algorithm 1 performs approximately 16 acceptance tests per job before finally rejecting a job.
In the last category, the table shows that for system utilization below 70% only approximately 1 acceptance test per aperiodic job is needed. When the utilization increases, global algorithm 1 performs on average up to 11.81 acceptance tests per aperiodic job and global algorithm 2 up to 26.87.

For completeness reasons, we also list the measured quickness values for this experiment for both global algorithms in Table 6.6[3]. The results are similar, but slightly worse than those observed in Experiment 1. Note that the results from Table 6.6 cannot be directly compared to those of the previous experiment listed in Table 6.4. The reason is

[3]We omit the results for EDF with background processing of aperiodic jobs and for partitioned slot shifting due to similarity to those of Experiment 1.

Type	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
accepted	global 1	1.01	1.02	1.03	1.06	1.11	1.18	1.38	2.00	2.82	2.74
	global 2	1.02	1.02	1.04	1.08	1.16	1.32	1.84	4.68	11.48	13.67
rejected	global 1	–	–	–	–	–	–	15.89	15.98	15.70	15.10
	global 2	–	–	–	–	–	–	–	32.00	32.00	32.00
total	global 1	1.01	1.02	1.03	1.06	1.11	1.18	1.38	2.18	6.84	11.81
	global 2	1.02	1.02	1.04	1.08	1.16	1.32	1.84	4.78	17.17	26.87

Table 6.5: *Experiment 2: mean value of the number of acceptance tests per aperiodic job for $U_{aperiodic}$ = 20%, DLX factor 2.*

$U_{aperiodic}$	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
10%	global 1	0.96	0.93	0.89	0.84	0.77	0.70	0.61	0.53	0.42	0.41
	global 2	0.95	0.92	0.88	0.83	0.77	0.69	0.59	0.50	0.40	0.42
20%	global 1	0.91	0.88	0.84	0.79	0.73	0.66	0.56	0.44	0.35	0.35
	global 2	0.91	0.87	0.83	0.78	0.72	0.63	0.53	0.41	0.32	0.37
50%	global 1	0.77	0.73	0.69	0.63	0.54	0.38	0.27	0.25	0.24	0.25
	global 2	0.75	0.70	0.64	0.57	0.45	0.29	0.21	0.19	0.21	0.26

Table 6.6: *Experiment 2: mean value of the quickness, DLX factor 2.*

that in this experiment slot shifting succeeds in adding more aperiodic jobs since there are more cores available to choose from. These additionally integrated aperiodic jobs worsen the mean quickness values slightly, as more ready jobs lead to higher response times. Tables D.16–D.18 in Appendix D list the results for different DLX factors.

6.2.3 Experiment 3: Balanced Utilization with SDL on 4/32 Cores

In Experiment 3a and b we use SDL and we repeat Experiment 1 and Experiment 2, respectively. The main focus is to compare the acceptance ratio and the quickness with those of the previous experiments to analyze how using SDL influences the behavior of the algorithms.

The resulting acceptance ratios for the scenarios of 10%, 20%, and 50% aperiodic job utilization differ by less than 0.2% from the acceptance ratios of Experiment 1 and Experiment 2, see Tables E.1–E.6 in Appendix E.

The same holds true when comparing the mean number of performed acceptance tests. Independent of the category and of the number of cores, Table 6.7 and 6.8 show only minuscule changes. We conclude that using SDL has no significant impact on the acceptance ratio of the aperiodic jobs and on the number of acceptance tests performed per aperiodic job.

Type	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
accepted	global 1	1.01	1.03	1.04	1.07	1.11	1.19	1.32	1.50	1.66	1.73
	global 2	1.02	1.03	1.05	1.08	1.15	1.26	1.47	1.81	2.15	2.37
rejected	global 1	4.00	4.00	4.00	4.00	3.99	3.99	3.99	3.98	3.97	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
total	global 1	1.01	1.03	1.04	1.07	1.12	1.21	1.42	1.91	2.78	3.57
	global 2	1.02	1.03	1.05	1.09	1.15	1.28	1.57	2.16	3.04	3.71

Table 6.7: *Experiment 3a: mean value of the number of acceptance tests per aperiodic job on 4 cores with SDL, $U_{aperiodic}$ = 20%.*

Type	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
accepted	global 1	1.01	1.03	1.04	1.07	1.11	1.19	1.38	2.00	2.82	2.73
	global 2	1.02	1.03	1.05	1.09	1.16	1.32	1.85	4.60	11.48	13.71
rejected	global 1	–	–	–	–	–	–	14.05	16.00	15.70	15.10
	global 2	–	–	–	–	–	–	–	32.00	32.00	32.00
total	global 1	1.01	1.03	1.04	1.07	1.11	1.19	1.38	2.18	6.84	11.82
	global 2	1.02	1.03	1.05	1.09	1.16	1.32	1.85	4.71	17.17	26.88

Table 6.8: *Experiment 3b: mean value of the number of acceptance tests per aperiodic job on 32 cores with SDL, $U_{aperiodic}$ = 20%.*

As previously in Experiment 1 and Experiment 2, we measure the the mean value of the quickness of the aperiodic jobs. We list the resulting quickness values in Appendix E: Tables E.7–E.9 list the results of Experiment 3a and Tables E.10–E.12 for Experiment 3b. Table 6.9 lists the improvement of using SDL when comparing the quickness of Experiment 1 with Experiment 3a. For a DLX factor of 2, the table shows that, apart

from minor fluctuations when the offline utilization is 0%, it is always beneficial to use SDL. With increasing utilization, SDL also improves the response times of the aperiodic jobs more. The maximum improvement is 28.6% for partitioned slot shifting, 26.6% for global algorithm 1, and 26.0% for global algorithm 2. The same effect is even stronger for a DLX factor of 5: Here, the maximum improvement is 155.4% for partitioned slot shifting, 135.2% for global algorithm 1, and 132.6% for global algorithm 2. The partitioned slot shifting algorithm achieves slightly higher gains using SDL. The reason is that some jobs that the partitioned algorithm rejects, are successfully integrated by the global algorithms. While this increases the acceptance ratio, the price is a higher mean value of the quickness of the aperiodic jobs: Finding a suitable core takes time, i.e., the response time of these jobs is higher and raises the average quickness.

DLX	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
2	part.	-3.1	2.9	7.8	13.8	20.1	24.5	27.0	28.6	26.7	24.6
	global 1	-3.1	2.8	7.6	13.6	19.8	24.4	26.5	26.6	21.9	18.8
	global 2	-3.1	2.7	7.6	13.7	19.9	23.9	25.7	26.0	22.2	19.0
5	part.	-2.3	2.8	7.2	12.9	21.0	33.9	57.1	98.0	155.4	148.8
	global 1	-2.3	2.8	7.2	12.9	21.0	33.9	57.5	98.5	135.2	108.3
	global 2	-2.3	2.8	7.2	12.9	21.0	33.9	57.3	97.1	132.6	107.3

Table 6.9: *Experiment 3a: improvement of the mean value of the quickness (in %), 4 cores,* $U_{aperiodic} = 20\%$.

Table 6.10 lists the improvement of using SDL when comparing the quickness of Experiment 2 with Experiment 3b. As expected, the table shows for the partitioned algorithm approximately the same results as Table 6.9. Apart from that, Table 6.10 shows the same trends on 32 cores as we have seen on 4 cores before. Besides a minor fluctuation at 0% offline utilization, using SDL is always beneficial to the response time of the aperiodic jobs. With increasing system utilization and with increasing DLX factor, this effect increases. A system featuring more cores allows for more delegations. While this is beneficial to the acceptance rate, it degrades the resulting average quickness of the aperiodic

DLX	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
2	part.	-3.1	2.9	7.7	13.8	20.0	24.8	27.2	28.2	26.8	25.0
	global 1	-3.1	2.8	7.4	13.1	18.7	23.7	28.6	27.3	13.4	9.6
	global 2	-3.1	2.8	7.5	13.6	19.8	24.3	26.0	24.1	15.1	10.9
5	part.	-2.3	2.8	7.2	12.9	21.0	33.9	57.2	98.3	156.4	149.4
	global 1	-2.3	2.8	7.2	12.9	21.0	33.9	57.1	98.1	97.6	58.8
	global 2	-2.3	2.8	7.2	12.9	21.0	33.9	57.5	97.4	86.4	45.3

Table 6.10: *Experiment 3b: improvement of the mean value of the quickness (in %), 32 cores,* $U_{aperiodic} = 20\%$.

jobs slightly compared to Experiment 3a which featured only 4 cores. In Appendix E, we list further results for other scenarios and other DLX factors in Tables E.13–E.18.

To summarize, Experiment 3 shows that using SDL has no measurable influence neither on the mean acceptance ratio nor on the average number of acceptance tests performed. The effect on the mean response time of the aperiodic jobs is as expected: For almost all tested parameters of the job set, using SDL improves the mean response time of the aperiodic jobs. Under high utilizations, the experiment shows that using SDL results in improvements on the mean quickness of the aperiodic jobs of more than 100% compared to normal slot shifting.

6.2.4 Experiment 4: Influence of the DLX Factor

Experiment 3 analyzed how using SDL influences the behavior of the slot shifting algorithms. With Experiment 4, we analyze the influence of the DLX factor on the quickness of the aperiodic jobs when using SDL.

For this experiment, the deadlines of the aperiodic jobs are set to 1.5, 2, 5, 10, 15, and 20 times the worst case execution time. The offline utilization is set to 50% and the utilization created by aperiodic jobs is set to 10%, 20%, and 50%, respectively.

Figures 6.17–6.19 show the resulting acceptance ratios for these aperiodic job utilizations, the exact values are listed in Figures F.1–F.3 in Appendix F. The larger the DLX factor becomes, the higher the acceptance ratio for all algorithms rises, since the aperiodic jobs become more flexible. EDF with background processing is always outperformed by the partitioned slot shifting algorithm, which in turn is always outperformed by its global counter parts.
The first two figures depict similar acceptance ratios: Since the total system utilization is comparably low, only small DLX factors limit the acceptance ratio. With increasing DLX factor, the acceptance ratio of all algorithms rapidly reaches 100%.
In Figure 6.19, the general trend is the same, but the higher total system utilization makes it much harder to achieve high acceptance ratios.

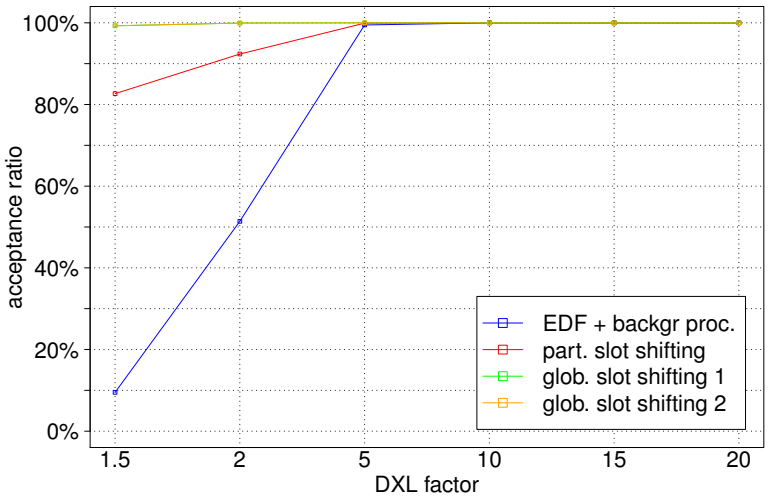


Figure 6.17: *Experiment 4: mean acceptance ratio of aperiodic tasks on 4 cores,* $U_{aperiodic}$ *= 10%.*

Figure 6.20 exhibits the resulting mean quickness of the aperiodic jobs for $U_{aperiodic}$ = 10% for the different algorithms with and without SDL. In general, there are three distinct curves visible: two for slot shifting with and without SDL, and one for EDF with background processing of aperiodic jobs. The latter curve unites with the curve of slot shifting without SDL for DLX factors of 5 and larger. For DLX factors of 1.5 and 2,

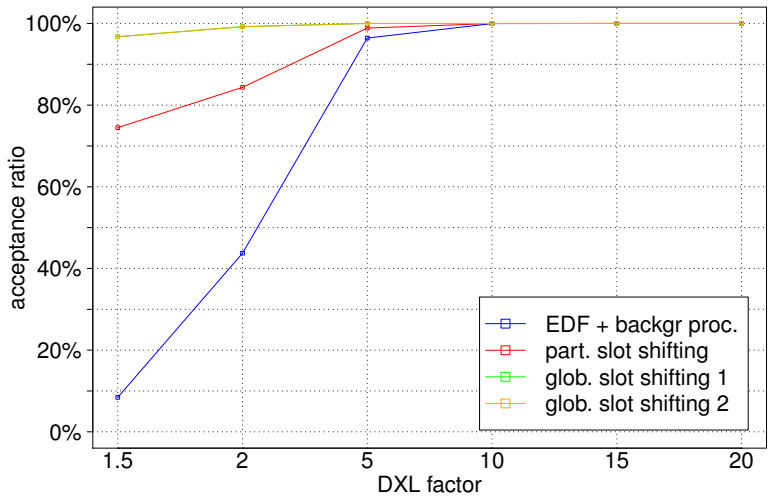


Figure 6.18: *Experiment 4: mean acceptance ratio of aperiodic tasks on 4 cores,* $U_{aperiodic} = 20\%$.

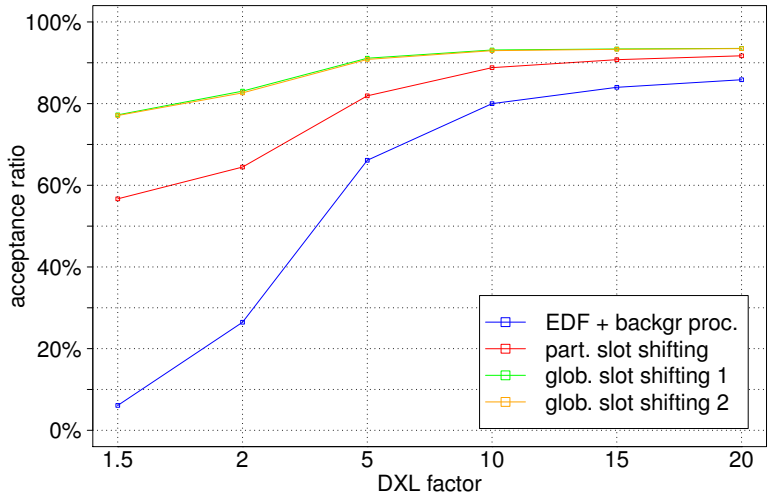


Figure 6.19: *Experiment 4: mean acceptance ratio of aperiodic tasks on 4 cores,* $U_{aperiodic} = 50\%$.

the quickness achieved with EDF with background processing is much lower than that of any other algorithm[4]. In each of the two distinct curves, all slot shifting algorithms show nearly identical behavior, with only minor differences between the different algorithms at small DLX factors. When the DLX factor becomes 5 and larger, all algorithms perform identically well in terms of quickness in their respective curve, again with small benefits

[4] Additionally, note that the resulting quickness of 0.50 for a DLX factor of 1.5 is only based on a few jobs, since EDF with background processing of aperiodic jobs only successfully executes about 8% of all aperiodic jobs.

when using SDL. The more the DLX factor increases, the more the mean quickness of the aperiodic jobs converges to 1.

Figure 6.21, which depicts the resulting mean quickness of the aperiodic jobs for $U_{aperiodic} = 20\%$, shows only marginal differences to the previous figure. The same general trends as in the previous figure are visible; the resulting quickness is in general slightly lower than before.

When the utilization created by aperiodic jobs becomes 50% (see Figure 6.22), the achieved quickness values of the distinct algorithms become much more separated than in the previous figures. With increasing DLX factor, the quickness drops for all algorithms compared to the previous figure. Note that with increasing DLX factor also the acceptance ratio increases (see Figure 6.19), i.e., there are more ready jobs which explains the decreased quickness. Again, EDF with background processing shows the worst resulting quickness values for DLX factors of 1.5 and 2. Interestingly, the quickness increases again for a DLX factor of 5 and stays then approximately constant. Among the slot shifting algorithms, partitioned slot shifting achieves the best quickness values, followed by global slot shifting algorithm 1 and 2, which both show only small differences. The better results for partitioned slot shifting were expected from the results of Experiment 3: The partitioned slot shifting algorithm rejects some aperiodic jobs (which leads to a lower acceptance ratio), whereas the global slot shifting algorithms try to send them to another core (which leads to their better acceptance ratio, but also decreases their quickness).
Another observation is that whenever SDL is used, the corresponding algorithm outperforms its "normal" counterpart.

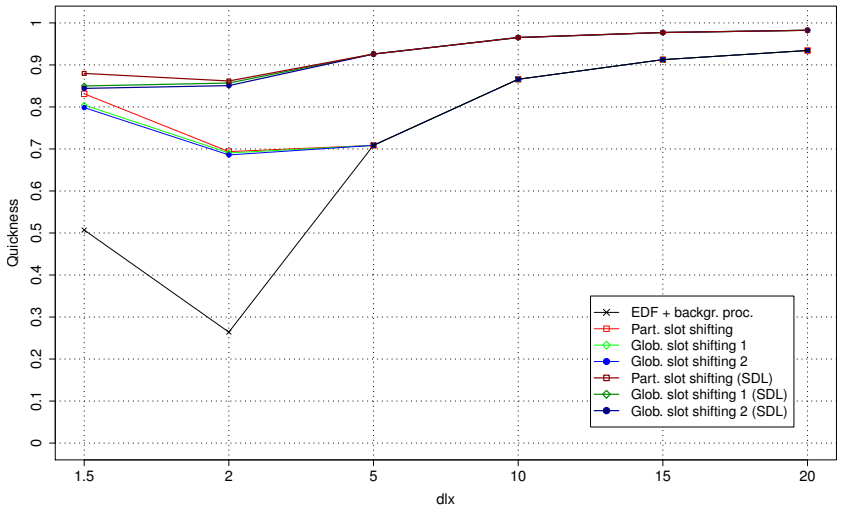


Figure 6.20: *Experiment 4: mean value of the quickness of aperiodic tasks on 4 cores,* $U_{aperiodic} = 10\%$.

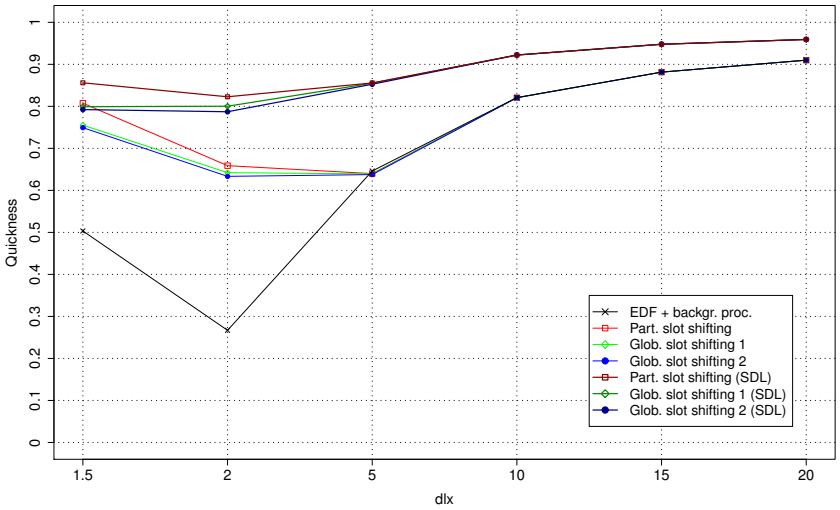


Figure 6.21: *Experiment 4: mean value of the quickness of aperiodic tasks on 4 cores,* $U_{aperiodic}$ *= 20%.*

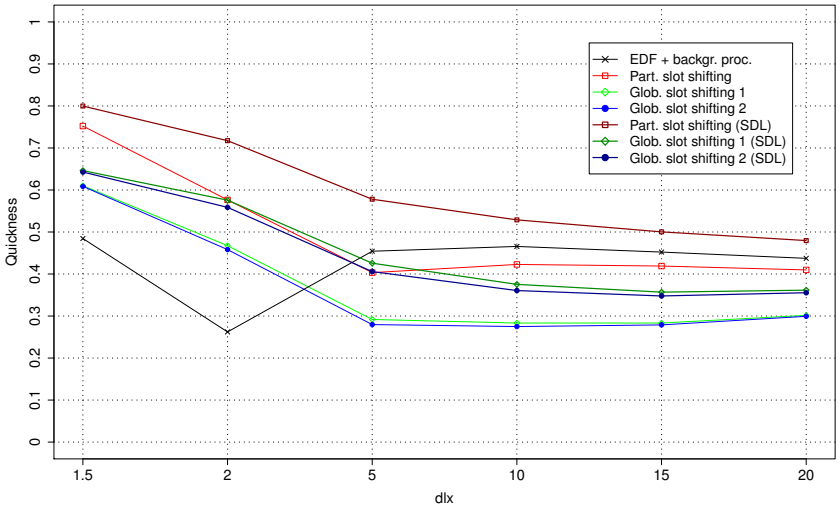


Figure 6.22: *Experiment 4: mean value of the quickness of aperiodic tasks on 4 cores,* $U_{aperiodic}$ *= 50%.*

6.2.5 Experiment 5: Dedicated Aperiodic Job Handling

This experiment analyzes how the global slot shifting algorithms handle different arrival patterns of aperiodic jobs. We perform the experiment on 4 cores and we use the same parameters for the offline guaranteed jobs as in Experiment 1 (see Table 6.1). We gradually change the arrival pattern from dedicated aperiodic job handling to a balanced arrival on all cores: In Experiment 5a, one core is dedicated to handle all aperiodic requests. In Experiment 5b, the aperiodic jobs arrive on two cores. In Experiment 5c, the aperiodic jobs are sent to all 4 cores but in a unbalanced-fashion as shown in Table 6.11. For comparison, the table also shows the distribution of the aperiodic jobs in the Experiment 1 (and with SDL: Experiment 3).

Experiment	U_{core1}	U_{core2}	U_{core3}	U_{core4}
Experiment 5a	200	0	0	0
Experiment 5b	100	100	0	0
Experiment 5c	80	60	40	20
Experiment 1/3	50	50	50	50

Table 6.11: *Parameters for Experiment 5; utilization created by aperiodic jobs (listed per core)*

Table 6.12 lists the resulting acceptance ratios for the global slot shifting algorithms. For clarity reasons, we only show the results for $U_{aperiodic} = 20\%$ and a DLX factor of 2. Results for other aperiodic job utilizations and other DLX factors are listed in Appendix G, see Tables G.2–G.19.

Table 6.12 also lists for comparison reasons the acceptance ratios of Experiment 1, which feature the same offline job parameters and a perfectly balanced arrival pattern of the aperiodic jobs. The results show that the global algorithms manage to handle different arrival patterns very effectively. Whether the aperiodic jobs arrive on all cores, or are handled by a dedicated core, the resulting acceptance ratios are approximately the same. The differences between the results of Experiment 5a, b, c, and Experiment 1

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 5a	global 1	100	100	100	99.98	99.89	99.38	96.46	83.58	52.16	17.05
	global 2	100	100	99.99	99.98	99.84	99.24	96.31	83.79	52.53	17.17
Exp. 5b	global 1	100	100	100	99.97	99.86	99.25	96.30	82.97	51.68	16.92
	global 2	100	100	100	99.97	99.82	99.15	96.22	83.29	52.00	17.03
Exp. 5c	global 1	100	100	100	99.97	99.85	99.26	96.27	83.02	51.78	16.88
	global 2	100	100	99.99	99.97	99.84	99.21	96.17	83.21	52.12	16.94
Exp. 1	global 1	100	100	100	99.98	99.87	99.25	96.08	83.28	51.35	17.10
	global 2	100	100	100	99.98	99.85	99.20	96.04	83.49	51.58	17.21

Table 6.12: *Experiment 5: measured mean acceptance ratio in %, $U_{aperiodic} = 20\%$, DLX factor 2.*

are only marginal.
Furthermore, the results confirm our findings of Experiment 3: The resulting acceptance ratios are independent whether or not SDL is applied (for the results of the acceptance ratio measurement for Experiment 5 with SDL see Table G.1 in Appendix G).

Table 6.13 lists for three different categories the mean number of acceptance tests for both global algorithms. We also measured the mean number of acceptance tests performed when SDL is used. As the results are very similar, we moved the results to Appendix G, see Table G.20. Further results without and with SDL for various other job parameters can be found in Tables G.21–G.47 and Tables G.48–G.74, respectively.
Dashes indicate that there was not a single job that was accounted for in this category, e.g., in Experiment 5a at 0% offline utilization both algorithms accepted all aperiodic

Mean Number of Acceptance Tests		Utilization Created by Offline Jobs 0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Mean Number of Aperiodic Jobs											
Exp. 5a	global 1	1.19	1.22	1.28	1.34	1.42	1.51	1.61	1.70	1.77	1.77
	global 2	1.21	1.26	1.34	1.43	1.53	1.66	1.84	2.10	2.34	2.45
Exp. 5b	global 1	1.06	1.07	1.11	1.15	1.23	1.32	1.44	1.59	1.71	1.75
	global 2	1.06	1.08	1.12	1.19	1.27	1.41	1.60	1.90	2.20	2.38
Exp. 5c	global 1	1.02	1.03	1.05	1.08	1.13	1.21	1.34	1.52	1.67	1.73
	global 2	1.03	1.04	1.06	1.11	1.19	1.32	1.52	1.87	2.21	2.42
Exp. 1	global 1	1.01	1.02	1.03	1.06	1.11	1.19	1.32	1.50	1.66	1.73
	global 2	1.02	1.03	1.04	1.09	1.15	1.28	1.50	1.84	2.17	2.41
Finally Rejected Aperiodic Jobs											
Exp. 5a	global 1	–	–	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.96
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
Exp. 5b	global 1	–	–	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
Exp. 5c	global 1	–	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
Exp. 1	global 1	4.00	4.00	4.00	4.00	3.99	3.99	3.99	3.98	3.97	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
All Aperiodic Jobs											
Exp. 5a	global 1	1.19	1.22	1.28	1.34	1.43	1.53	1.69	2.08	2.82	3.58
	global 2	1.21	1.26	1.34	1.43	1.54	1.68	1.92	2.41	3.12	3.73
Exp. 5b	global 1	1.06	1.07	1.11	1.16	1.23	1.34	1.54	1.99	2.80	3.58
	global 2	1.06	1.08	1.12	1.19	1.27	1.43	1.69	2.25	3.06	3.72
Exp. 5c	global 1	1.02	1.03	1.05	1.08	1.14	1.23	1.43	1.93	2.77	3.58
	global 2	1.03	1.04	1.06	1.11	1.19	1.34	1.62	2.22	3.06	3.73
Exp. 1	global 1	1.01	1.02	1.03	1.06	1.11	1.21	1.42	1.91	2.78	3.57
	global 2	1.02	1.03	1.04	1.09	1.16	1.30	1.60	2.20	3.05	3.72

Table 6.13: *Experiment 5: mean number of acceptance tests per aperiodic job, $U_{aperiodic}$ = 20%, DLX factor 2.*

jobs, thus there was no finally rejected job. Although the table lists, e.g., for Experiment 1 at 0% offline utilization a value of 4.00 for global algorithm 2 this value is only based on a single job that was rejected out of all 173504 aperiodic jobs.
The following observations can be made: For both algorithms, the mean number of acceptance tests performed for finally rejected jobs is approximately the same for all arrival patterns that we tested. The results for both categories, accepted aperiodic jobs and all aperiodic jobs, show a trend: When one single dedicated core receives all aperiodic jobs, on average more acceptance tests need to be performed to integrate the aperiodic jobs. The more the arrival pattern changes towards a perfectly balanced pattern as in Experiment 1, the less acceptance tests need to be performed.

As in the previous experiment, we also measure the resulting mean quickness for the aperiodic jobs for normal slot shifting and with SDL, see Table 6.14 and 6.15, respectively. Further results without and with SDL for various job parameters can be found in Appendix G in Tables G.75–G.83 and in Tables G.84–G.92, respectively.

Mean Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 5a	global 1	0.72	0.70	0.68	0.66	0.63	0.59	0.52	0.46	0.44	0.48
	global 2	0.71	0.69	0.66	0.63	0.61	0.56	0.50	0.44	0.43	0.48
Exp. 5b	global 1	0.83	0.79	0.76	0.72	0.67	0.61	0.53	0.46	0.44	0.48
	global 2	0.83	0.79	0.75	0.71	0.66	0.60	0.52	0.45	0.43	0.48
Exp. 5c	global 1	0.90	0.86	0.82	0.77	0.71	0.64	0.55	0.46	0.44	0.48
	global 2	0.89	0.86	0.82	0.76	0.70	0.63	0.54	0.46	0.44	0.48
Exp. 1	global 1	0.91	0.88	0.84	0.78	0.72	0.65	0.55	0.47	0.44	0.48
	global 2	0.91	0.87	0.83	0.78	0.71	0.64	0.54	0.46	0.43	0.48

Table 6.14: *Experiment 5: mean value of the quickness, $U_{aperiodic}$ = 20%, DLX factor 2.*

Mean Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 5a	global 1	0.69	0.72	0.73	0.74	0.75	0.73	0.66	0.58	0.53	0.57
	global 2	0.67	0.71	0.72	0.72	0.72	0.69	0.62	0.55	0.53	0.57
Exp. 5b	global 1	0.79	0.82	0.82	0.82	0.80	0.76	0.68	0.58	0.53	0.57
	global 2	0.79	0.81	0.81	0.81	0.79	0.74	0.66	0.57	0.53	0.57
Exp. 5c	global 1	0.87	0.89	0.88	0.88	0.85	0.79	0.69	0.59	0.54	0.57
	global 2	0.86	0.88	0.88	0.87	0.84	0.78	0.67	0.57	0.53	0.57
Exp. 3	global 1	0.88	0.90	0.90	0.89	0.87	0.80	0.69	0.59	0.53	0.58
	global 2	0.88	0.90	0.89	0.88	0.86	0.79	0.67	0.58	0.53	0.58

Table 6.15: *Experiment 5 with SDL: mean value of the quickness, $U_{aperiodic}$ = 20%, DLX factor 2.*

When comparing both algorithms, it becomes visible that global algorithm 1 outperforms algorithm 2 slightly in terms of response time. This effect weakens and then

finally disappears with increasing system utilization. In general, with increasing system utilization the quickness of the aperiodic jobs decreases, i.e., the average response time increases. Against this trend, at 90% system utilization the response time decreases again.
The dedicated handling of aperiodic jobs at a single core results in larger response times. The more the aperiodic job arrivals get distributed and balanced on the cores of the system, the more the response times of the aperiodic jobs improve.

6.3 Discussion

In section 6.2, we presented the results of the different experiments that we conducted on Elwetritsch, the high performance cluster of the University of Kaiserslautern. We analyzed the effectiveness of the different slot shifting algorithms in terms of: acceptance ratio of the aperiodic jobs, average number of performed acceptance tests, and resulting responsiveness of the aperiodic jobs. In this section, we summarize and discuss the most important findings and insights gained from the experiments.

With Experiment 1, we compared the acceptance ratio of the different slot shifting versions. Additionally, we run EDF with background processing of the same job sets. The latter algorithm just serves as a comparison metric. As expected, the algorithm is always outperformed by partitioned slot shifting, which in turn is surpassed by both global slot shifting algorithms. The acceptance ratio of the spare-capacity-based global slot shifting algorithm is always slightly smaller than that of the negotiation-based global slot shifting algorithm. In this experiment, we found the upper and lower bounds on the 95% confidence intervals of the mean acceptance ratio to be very tight, which gives strong evidence on the reliability of our results.
Using global instead of partitioned slot shifting improves the acceptance ratio by up to approximately 30%—depending on the utilization created by online and offline jobs. We obtained similar results for both global algorithms.

For offline utilizations smaller than 50%, both global algorithms perform approximately 1 acceptance test per aperiodic job on average. When the utilization created by the offline jobs increases further, this number increases towards 4, with global algorithm 2 performing slightly more acceptance tests than global algorithm 1. Nevertheless, note that the runtime measurements conducted in section 5.3 showed that global algorithm 1 generally runs slower than global algorithm 2.

To compare the absolute response times of the aperiodic jobs, we introduced a new normalized metric: the quickness. In general, the experiment shows that slot shifting always achieves better average responsiveness of the aperiodic jobs than EDF in which aperiodic jobs are run when the processor is idle. Independent of the algorithm used, most aperiodic jobs feature a very short response time, if the utilization created by the offline jobs is low. With increasing offine utilization, the response times of the aperiodic jobs grow and become more distributed. At $U_{offline} = 90\%$, approximately half of the aperiodic jobs feature the maximum response time and only 1.3% the minimum response time, when using EDF and processing the aperiodic jobs in the background. When using partitioned slot shifting, 5.8% of the aperiodic jobs feature the maximum response time and 15.8% the minimum response time. Using the global versions of slot shifting shows that 7.8% of the aperiodic jobs feature the maximum response time and 4.2% the minimum response time.
At first sight, this seems to indicate that partitioned slot shifting performs better than its global counter parts. When we also consider the absolute numbers and the resulting acceptance ratio, it turns out that global slot shifting offers better results: The global algorithms manage to accept nearly twice as many aperiodic jobs. The overall higher

acceptance ratio results in more processor utilization, i.e., more jobs interfere with each other and thus, the responsiveness decreases for some of them.

In Experiment 2, we repeat Experiment 1 on 32 cores. The experiment shows up to 30% increase of the acceptance ratio favoring the negotiation-based global algorithm more than the spare-capacity-based global algorithm. It also shows that the latter is able to accept approximately the same number of aperiodic jobs performing much less acceptance tests than the former algorithm. The results further prove the efficiency of the implementation, as the negotiation-based global algorithm—even under high system utilization—on average does not perform as many acceptance tests for the aperiodic jobs as there are cores in the system.

The responsiveness of the aperiodic jobs is in this experiment for both global algorithms similar, but slightly worse than those observed in Experiment 1. In Experiment 2, slot shifting succeeds in adding more aperiodic jobs since there are more cores available to choose from. These additionally integrated aperiodic jobs worsen the results slightly, as more ready jobs lead to higher response times.

In Experiment 3a and b, we repeated Experiment 1 and Experiment 2 using SDL on 4 and 32 cores, respectively. From the results of the experiments, we conclude that using SDL has no significant impact on the acceptance ratio of the aperiodic jobs and on the number of acceptance tests performed per aperiodic job.

However, the effect on the average response time of the aperiodic jobs is as expected: For almost all tested parameters, using SDL improves the response time of the aperiodic jobs. Under high utilizations, the experiment shows that using SDL results in improvements on the average quickness of the aperiodic jobs of more than 100% compared to normal slot shifting.

In Experiment 4 we fixed $U_{offline}$ and varied the DLX factor to analyze the influence on the behavior of the different slot shifting algorithms. As expected, the overall resulting acceptance ratio increases with increasing DLX factors—independent of the utilization created by the aperiodic jobs. With $U_{aperiodic} = 10\%$ (and with $U_{aperiodic} = 20\%$), the average quickness of all slot shifting algorithms is similarly close to the optimum and as one would expect slightly improving with increasing DLX factor. With $U_{aperiodic} = 50\%$, the overall acceptance ratio is smaller as the processor is more loaded. This case is much more demanding, hence, the responsiveness of the aperiodic jobs is lower and deteriorating as the DLX factor (and thus the acceptance ratio) increases. In other words, with increasing number of jobs in the system, aperiodic jobs take longer to complete execution, thus their responsiveness deteriorates. Furthermore, the resulting quickness curves for the individual algorithms are much more distinct in this case than before. So we conclude that with relatively low utilizations created by aperiodic jobs (10% and 20%) increasing the DLX factor, i.e., the length of the deadline of the aperiodic jobs relative to their WCET, is beneficial for the acceptance ratio and leads to good responsiveness. When the load created by the aperiodic jobs becomes higher, increasing the DLX factor still improves the acceptance ratio. The responsiveness, however, does not

follow this trend anymore and deteriorates.

In the last experiment, we saw that the global algorithms manage to handle different arrival patterns very effectively. Independent of the actual arrival pattern, the resulting acceptance ratios are approximately the same. The more the arrival pattern changes from a perfectly balanced pattern towards dedicated handling on a single core, the more acceptance tests are performed. Furthermore, handling the aperiodic jobs at a single dedicated core results in larger mean response times of the jobs. The more the aperiodic job arrivals get distributed and balanced on the cores of the system, the more the response time of the aperiodic jobs improves. In general, with increasing system utilization the average response time of the aperiodic jobs increases. Against this trend, at 90% system utilization the response time decreases again.

Chapter 7

Resource Management for Linux

Today's multicore-based embedded systems are executing many applications in parallel. As already explained in section 1.3.7, the access to the limited resources of such platforms must be managed to cope with contention, resolve constraints, and avoid starvation of individual applications. The primary goal of resource management is to ensure resource availability to applications such that the overall QoS requirements are fulfilled. In order to achieve this, the resource management must actively distribute the resources of the platform among the applications. Therefore, it must encapsulate and isolate the individual applications to some degree from each other during execution.

Under adaptive resource management, applications are assumed to offer several modes of execution with different resource requirements and providing different QoS. Adaptive resource management monitors the resource consumption of the applications and reclaims unused resources. In time periods when the system suffers from overload, adaptive resource management requests mode changes of the applications and reassigns the resources to globally maximize the system performance and hence, the user perceived QoS.

Examples of adaptive applications include multimedia applications that implement multi-version algorithms with different resource needs. This is often used to offer a low quality and a high quality playback functionality. There also exist so called anytime algorithms which always provide a solution to a given problem, independent of their elapsed runtime. Precision and quality of the calculated solution, however, improve as the algorithm spends more time solving the problem. Newton's iterative root finding algorithm [83] is an example for an anytime algorithm. If an application offers only a single mode of operation, then the resource management always has to fulfill its entire resource requirement. Hence, without any adaptive applications in the system, adaptive resource management reduces to static distribution of resources.

In this chapter, we use a generic adaptive resource management framework to show that slot shifting offers a feasible solution to provide deterministic guarantees to applications while allowing for runtime flexibility. Our approach is to implement a slot-shifting-based logic into a resource management framework as a proof of concept and to show the technical feasibility.

The rest of the chapter is structured as follows: First, we give an overview of the resource management framework: We describe its architecture and give an overview of

its working principles. Then, there follows a section which illustrates the capabilities of ACTORS introducing different applications and performing experiments. After that, we present our approach to integrate a slot-shifting-based logic into the framework. Then, we discuss the challenges and the solutions that we found and evaluate our approach. We conclude the chapter with a discussion of our findings.

7.1 Overview

The ACTORS framework [72] is an adaptive generic resource management framework for multicore platforms. The acronym ACTORS stands for Adaptivity and Control of Resources in Embedded Systems. ACTORS addresses resource-constrained embedded multicore platforms with high requirements on adaptivity.

7.1.1 Concepts

Figure 7.1 depicts an overview of the ACTORS resource management framework. The bottom of the figure shows the physical platform, usually consisting of one or multiple multicore processors. ACTORS employs the concept of a *virtual processors* (VP) to manage the applications, see the center of the figure. A process running in a VP is temporally isolated from other applications in the system. From the application's points of view, its processes execute individually or in a group as the only process(es) on their private but slower processor, hence the name virtual processor. In the ACTORS framework, applications are assigned to one or multiple VPs that altogether form a *virtual platform*. A virtual platform is an abstraction that provides isolation between running applications. The concept is not new: Nesbit et al. introduced in [84] virtual private machines to provide an abstract view of the available physical resources of the platform.

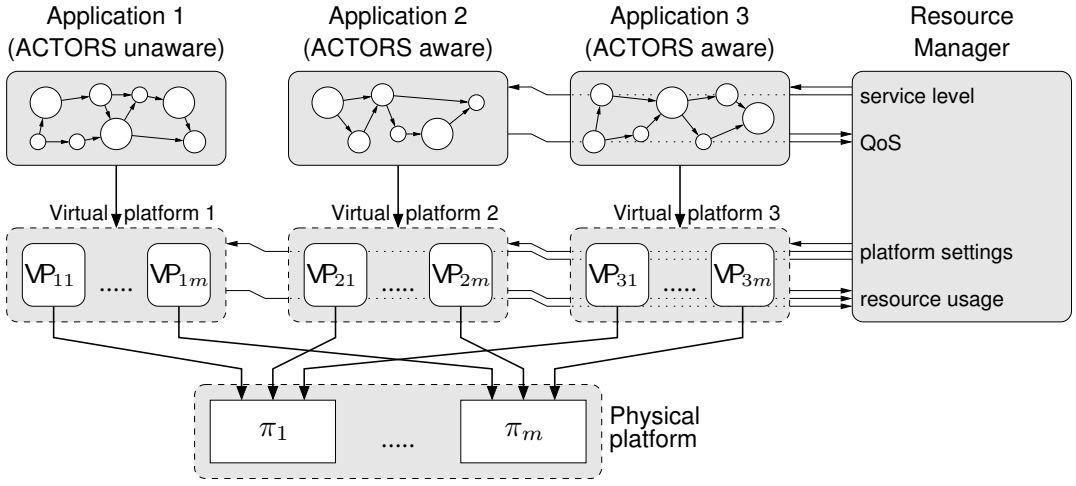


Figure 7.1: *Overview of the ACTORS resource management framework [3].*

A key idea of ACTORS is to automatically assign one or multiple VPs to every application and dynamically adjust these VPs according to the application's resource needs. The entity that is in charge of this decision making process is the resource manager (RM), depicted on the right side of Figure 7.1. The RM instructs the operating system to set up (or destroy) the virtual platforms. Also, the RM manages the parameters of the VPs and monitors the applications running in the virtual platforms at runtime of the system.

In the ACTORS framework, applications are assumed to feature different modes of execution with different resource requirements. In order to reflect this, ACTORS employs the concept of a service level (SL). An application features at least one, usually multiple SLs. Each SL specifies the resource requirements and the QoS provided by the application when run in this particular SL. To express resource needs, a SL specifies a (minimum) number of VPs. Further, ACTORS uses a *bounded delay abstraction*, the *alpha delta model* [85] to express resource needs for the individual VPs. In this model, the bandwidth α represents the amount of resources required periodically for the application to run. The delay Δ expresses the worst case service delay, i.e., the maximum tolerable time interval that the application can endure before it needs to be scheduled again. This abstraction is also commonly used in other fields such as networking [86] and disk scheduling [87].

Whenever an application starts up, it registers itself with the RM thus announcing its different execution modes and the associated resource requirements. Additionally, every application features an *importance* parameter[1] which allows the user to rank different applications, i.e., prompt the RM to favor them. After registration, the RM determines based on the importance of all applications and the resource availability of the overall system the SL of each individual application.

Hence, the arrival of new applications potentially triggers reassignments of SLs of all applications in the system and thus reallocation of resources. The RM monitors the virtual platforms' resource consumptions to react to fluctuations and to reclaim unused resources. Additionally, the applications inform the RM about their state, i.e., whether they run smoothly within they virtual platform or whether they suffer under resource constraints. To achieve this, applications use the *happiness* parameter (not shown in Figure 7.1) to provide feedback to the RM. For example a video decoder application informs the RM with this happiness parameter whether it is able to keep the desired frame rate, thus providing the QoS associated with the selected SL. The standard version of the logic uses a combination of feed-forward and feedback strategies to modify resource allocation at runtime. Informally speaking, the RM dynamically adjusts virtual platform parameters at runtime based on the measured resource consumption and the feedback from the applications. When applications finish, they unregister with the RM. Thereupon, the RM reassesses the SL assignments of all applications.

[1] Figure 7.1 is simplified and omits some abstractions such as importance and happiness.

7.1.2 Implementation

The ACTORS framework runs on top of a Linux operating system whose kernel has been patched to implement a new scheduling class called *SCHED_EDF*. This scheduling class enriches Linux and make it suitable for (soft) real-time scheduling. Most importantly, the patch introduces new process parameters: Processes feature a worst case execution time and a deadline. Moreover, SCHED_EDF implements an EDF-based scheduler, hence its name. Additionally, SCHED_EDF features a CBS-based server mechanism which is instrumented in the ACTORS framework to implement the VPs. The parameters α and Δ of a periodic server with period P and budget Q are calculated as follows:

$$\alpha = \frac{Q}{P} \qquad \Delta = 2(P - Q) \tag{7.1}$$

In order to interface with SCHED_EDF, cgroups, a feature of the Linux kernel, is employed. Cgroups establishes a virtual file system which is used to create and destroy VPs, specify their reservation parameters, and monitor the resource consumption of the individual VPs.

On top of SCHED_EDF Linux runs the resource manager, realized as a multi-threaded C++ application that executes with superuser privileges. To communicate with applications, the RM uses D-BUS [88], one of Linux' standard open-source inter-process communication protocols. The RM is a modular program which facilitates easy exchange of its internal logic: Up to date, six different implementations of the RM logic have been devised. Each version implements a different strategy to assign and manage the resources of the platform. Strategies involve, e.g., integer linear programming (ILP) or are based on the gravitational task model. The standard logic uses whenever an application starts or terminates an ILP formulation to assign the individual SLs and map the VPs to the cores. To solve the hereby established complex equation system, the standard logic uses the GNU Linear Programming Kit (GLPK) library [89]. At runtime of the system, this logic periodically monitors all VPs and dynamically adjusts the VP parameters to reclaim unused resources, as already mentioned.

7.1.3 Applications

To illustrate the capabilities of the ACTORS approach, different applications have been created. One example application controls an industrial robot whose claw holds to an inverted pendulum. Depending on the available computational resources, the application manages to balance and stabilize the inverted pendulum very efficiently, i.e., without much oscillation of the pendulum. In lower service levels, i.e., with reduced resources, the pendulum shows more oscillations around the set point and may even become unstable.

Similarly, another application has been implemented that controls a servo motor which regulates the tilt angle of a beam. On top of this beam, a ball rolls back and forth. As the previous application, this application stabilizes the ball on the beam using feedback control.

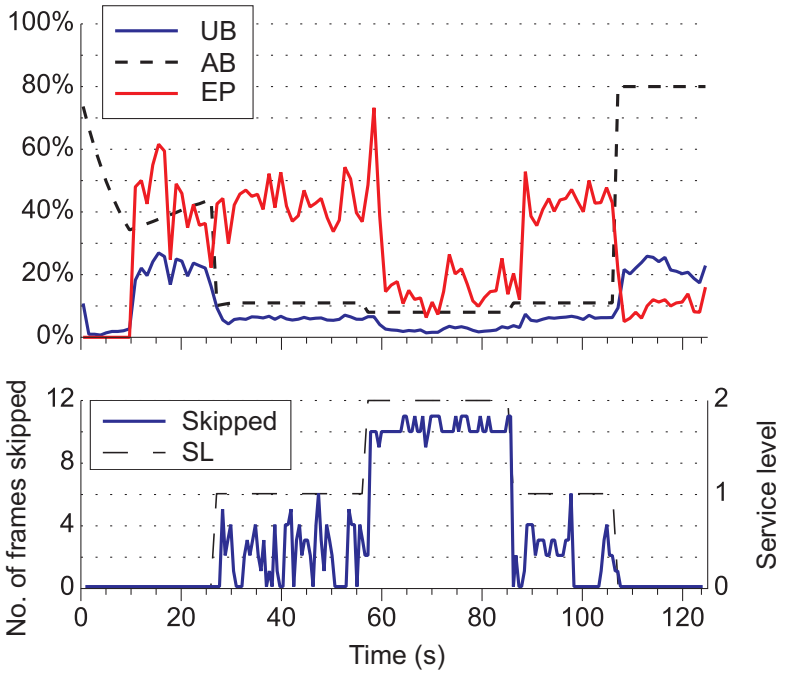


Figure 7.2: *Bandwidth and service level variation for the MPEG decoder. The figure gives an overview of the client's SL, used and assigned bandwidth, and exhaustion percentage of the virtual platform. It also shows the number of skipped frames per second in the adapted stream. As the SL is reduced, the number of skipped frames increases.*

Various experiments have been run to demonstrate that the ACTORS framework is capable to handle multiple of these ball and beam control applications together with the inverted pendulum and a multitude of other applications running concurrently on the same multicore platform.

To show that ACTORS handles also applications with highly varying bandwidth requirements, we designed an adaptive MPEG decoder. It is implemented as a distributed application consisting of an adaptive video server [90] and a playout client running on different machines connected over a wireless network. The video client is a modified VLC player that provides three distinct service levels with an overall bandwidth requirement of 80%, 11%, and 8%, respectively.

Figure 7.2 shows the MPEG decoder's runtime behavior during an experiment. The top of the figure reports the client application's assigned and used bandwidth (AB and UB), together with the exhaustion percentage (EP) of its virtual platform. Initially, the client is the only application running on the platform, thus it runs at the highest service level (0). The figure shows that the feedback mechanism dynamically adjusts the assigned bandwidth, because the assigned resources exceed the used resources by far. After about 25 seconds, another application is activated. The RM informs the video client to switch to a lower service level (1) and instructs the operating system to shrink the corresponding reservation. In this case, some frames are skipped, as shown at the bottom of Figure 7.2. After 50 seconds, a third application starts up, so the RM enforces even stricter constraints on the video client. This pushes the client to service level 2

which provides the lowest QoS. At time 90, the RM sets the SL back to 1 because the last application finished execution. Finally, at time 115, the initial condition is restored.

7.2 Slot Shifting

In the following, we describe our approach to use slot shifting to provide deterministic guarantees to applications and while allowing for runtime flexibility in the ACTORS resource management framework. First, we illustrate the underlying concepts of our approach. Then, we detail the implementation specifics. Finally, there is a section that brings the experimental evaluation and proves the technical feasibility of our approach.

7.2.1 Concepts and Implementation

Our approach realizes a partitioned slot shifting logic for the ACTORS framework. The design consists of two parts to separate the creation from the scheduling of the jobs: First, a job-starter tool that is in charge of starting the offline jobs with the parameters specified in the offline scheduling table. Additionally, this tool starts up the aperiodic jobs. It registers the both types of jobs with the RM, hereby informing the RM about their parameters and the associated Linux process IDs. Second, the RM logic which implements the slot shifting algorithm. On each core of the system, there exists an independent scheduler thread. These schedulers rely on a globally synchronized time to carry out the slot shifting algorithm based on the offline scheduling table for each core. They also perform the acceptance test and guarantee algorithm for the aperiodic jobs.

There are some conceptional differences between our desiign of the slot shifting logic for ACTORS and the slot shifting algorithm as presented in chapter 2: First, the original algorithm runs only once through the table and does not clearly specify how to proceed: While traversing towards the end of the offline scheduling table, slot shifting modifies the table. Thus, the table cannot be directly used to continue execution in a cyclic fashion. In the ACTORS framework however, the system must keep running forever. Thus, the table must be periodically prolonged as the time progresses. By extending the table, the logic is always kept in a defined state and continues execution in a cyclic fashion. Another reason for this extension of the table is that aperiodic jobs with deadlines after the end of the table can be handled.

A second difference comes with the notion of a slot. According to the slot shifting algorithm exactly one job may execute per core and per slot. This strict concept must be weakened in the ACTORS framework for several reasons. SCHED_EDF Linux employs constant bandwidth servers to setup individual reservations and to enforce temporal isolation among the processes. If only a single job per core in its private reservation is allowed to run, then the system would fail. The Linux operating system and its graphical user interface heavily rely on a set of daemon and server processes. These processes must be allowed to execute to offer essential system services and to perform background activities. Otherwise, interactivity with the user would cease as mouse, keyboard, and graphical front-end would freeze. Additionally, the RM and the

slot shifting schedulers on each core need to get a share of the processor time to react to incoming aperiodic jobs and to perform the slot shifting algorithm.

The RM consists of multiple threads: The main thread first parses the offline scheduling table and, after performing a basic check on the table data, initializes all data structures. Therefore, it creates two copies of the table data: First, a working copy, which is used and updated at runtime to perform the slot shifting algorithm, and second, an unmodified backup copy which is used at runtime to periodically extend the table as time progresses. The next step is to double the length of the working copy of the offline table to enable to logic to handle aperiodic jobs with deadlines up to twice the length of the offline scheduling table. After that, the main thread creates for every core in the system an individual "logic thread" which is in charge of making all slot shifting related decisions for the particular core. Then, the main thread creates two distinct CBS reservations on every core: One for the actual job to be scheduled within the slots and one for the RM threads. The exact settings of the reservations are discussed in the next section.

Figure 7.3 shows the simplified control flow implemented by the slot shifting logic threads. First, if there has been any job scheduled on the same core in the previous slot, then the logic sends a signal (SIGSTSP) to "freeze" the workload process, i.e., to remove it from the ready queue of the Linux scheduler. In the next step, the logic checks whether any aperiodic jobs have arrived during the last slot. For every arrived aperiodic job, it performs an acceptance test and based on the result rejects or guarantees the aperiodic job. Independent whether or not aperiodic jobs arrived or not, in the next step, the logic updates the ready lists of aperiodic and offline jobs. Then, based on the available spare capacity of the current interval and the deadlines of the ready jobs, the logic selects the next job to be scheduled. Finally, the logic schedules the selected job by updating the corresponding reservation and sending a signal (SIGCONT) to the job.

After the logic threads have been started, the main thread of the RM is in charge of managing the global time base and of periodically waking up the logic threads, which is done using mutexes and the pthread_cond_timedwait() function. Furthermore, it notifies the job-starter tool when a new slot starts, such that this tool synchronizes to the same global time base. As a result, the job-starter tool is able to reproducibly start aperiodic jobs in pre-defined slots.

The RM's standard interface D-BUS to the applications has multiple issues—see discussion in section 7.3—and has proven to be too slow for our implementation. Therefore, we implemented a faster interface to register jobs with the RM. This interface is based on shared memory and the low-level kill() system call to exchange signals between the processes. The interface intentionally does not use handshakes between the RM and the job starter tool. As a result, the RM cannot be delayed from the job starter tool. The theoretical drawback of the lack of handshakes is that if this interface is slowed down for some reason, then aperiodic jobs might arrive one slot delayed at the RM without the job starter tool noticing it. Our implementation monitors the time required to send

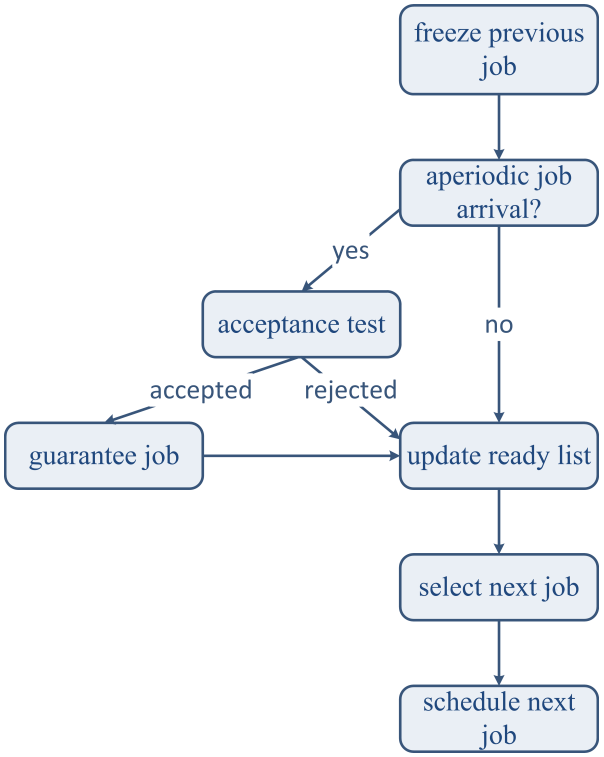


Figure 7.3: *Overview: simplified control flow of slot shifting logic thread for the ACTORS RM.*

and process aperiodic jobs to the RM and such theoretically possible overly long delays have never been observed.

Whenever the job-starter tool sends new aperiodic jobs to the RM, an asynchronous handler which processes the job parameters is triggered. Per job that is sent to the RM, 48 bytes of shared memory are required to store the job parameters. The handler makes alternate use of two queues to store the job parameters. This avoids race conditions and accidentally overwriting the data currently used by the logic to process previous slot's aperiodic jobs. In other words, our implementation intrinsically ensures mutually exclusive access to the data. Since our implementation does not employ blocking directives such as mutexes, it is free of additional blocking delays.

A detailed slot and overhead definition is given by Figure 7.4. As in the theoretical slot concept presented by Figure 2.1 in section 2.1, a slot consists generally of the time t_S needed to make a scheduling decision and the time t_E, in which the workload job is scheduled. After the event signaling the start of a new slot occurred, there exists a small delay t_d before the code of the slot shifting logic thread is scheduled by Linux. For this reason, the beginning and the end of the complete slot is slightly shifted in the figure. Within t_S, a certain time interval t_{VP} is required by our implementation to enact the scheduling decision, i.e., to change the scheduling class of the job to SCHED_EDF if not done before, and to update the budget and period parameters of the job and the VP that hosts this job, if needed.

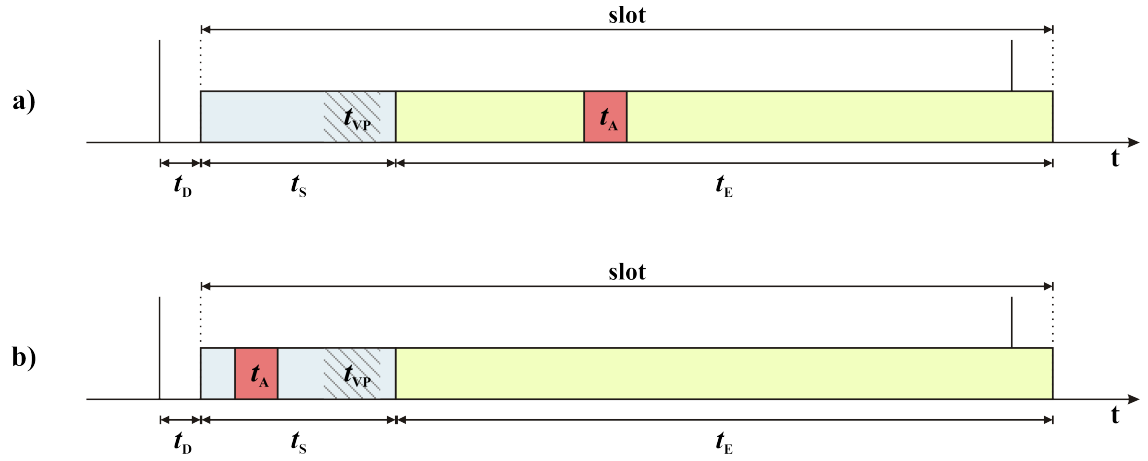


Figure 7.4: *Detailed slot and overhead definition: The figure depicts the time needed to make a scheduling decision t_S including the time to perform the VP management t_{VP} and also shows the job execution time t_E. Additionally, the delay t_D between the timer event and the actual start of the slot shifting code is illustrated. At any time, one or multiple aperiodic jobs can arrive and cause interference t_A. The top of the figure shows interference during t_E; the bottom of the figure shows the same interference during t_S.*

Additionally, the asynchronous handler can be triggered at any moment in time and will then execute for t_A units of time. As depicted in Figure 7.4a), when this handler is triggered during the execution of the actual workload job, a small fraction of the time interval t_E is stolen to process the job parameters of newly arriving aperiodic jobs. During the implementation and testing of our approach, we observed that due to the efficiency of our code the handler usually fires earlier, i.e., in the middle of the execution of the logic thread, as shown in Figure 7.4b).

7.2.2 Experimental Evaluation

For the experimental evaluation of our slot shifting implementation for ACTORS, we used a standard notebook that runs the 32-bit version of Linux 2.6.33 SCHED_EDF. The notebook we run our implementation on features an Intel Core 2 Duo P8700 processor and 4GB of RAM. During the experiment, we set both cores to operate at maximum frequency of 2.53GHz.

We created and annotated an offline scheduling table of 250 slots length for 2 cores, using the tools described in section 5.2.1. Table 7.1 lists the settings used to create the offline scheduling table.

The CBS reservation for the workload job on each core (i.e. for t_E) features a bandwidth of 80% and a period of 100ms. While the individual threads that execute the logic (i.e., for t_S) reside in a CBS with 15% bandwidth and a period of 5ms. The reason for this is that we found a CBS period equal to the slot length to be inadequate for the logic reservations: The resulting scheduling times t_S were unacceptably high. The explanations for this is that the VP that hosts of the logic thread, the VP that hosts

Parameter	Offline Tasks	Aperiodic Tasks
worst case execution time	$1-15$	$10-15$
period	$15-30$	–
deadline	$15-30$	$2*C$
resulting U per core	50%	20%
max. abs. allowed deviation from target U	<1%	

Table 7.1: *Overview of the task set parameters, DLX factor 2.*

the workload job, and the wake-up events triggered by the main thread are not synchronized, since SCHED_EDF offers no means to synchronize one or multiple VPs to internal or external events. Furthermore, the RM main thread and the logic thread on core 0 run in the same VP. Thus, it can happen that the RM main thread consumed some fraction of the VP's budget and the remaining fraction is not sufficient for the logic thread to finish its execution. For this reason, the slot shifting logic needs multiple server executions to finish. By proportionally decreasing the budget and the period of the VP that hosts the logic thread, we could limit this effect to some extent.

We run aforementioned annotated offline scheduling table for 5000 slots with the slot length set to 100ms. Table 7.2 lists the observed runtimes for the slot shifting code, the time to run the jobs, and the resulting overall slot lengths. Furthermore, the table lists the time required to perform the VP management, the delay induced by the operating system before it executes the slot shifting code, and the interference caused by the signal handler routine that is in charge of new aperiodic job arrivals.

Time	Minimum	Mean	Maximum
slot length ($t_S + t_E$)	99626	100000.00	100486
scheduling time (t_S)	18	272.09	19707
job runtime (t_E)	79978	99727.90	100439
VP management (t_{VP})	6	312.28	19647
start delay (t_D)	33	272.02	658
aperiodic job arrival (t_A)	2	3.17	23

Table 7.2: *Measured runtimes in microseconds, based on 5000 slots on 2 cores with the slot length set to 100ms.*

The table shows that our implementation maintains the slot length on average precisely. The minimum observed slot length was only 0.37% too short and the maximum slot length was about 0.49% too long.

Furthermore, the table lists minimum and mean scheduling times of 18 and approximately 272μs. However, the maximum observed scheduling time of 19707μs is quite large. One explanation for this effect can be found when looking at the measured VP management times t_{VP}, which show similar maximum values. In this experiment and also in other experiments that we performed, we realized that updating the virtual file

system of cgroups is a slow operation. Adding a process IDs to an existing VP takes always a few milliseconds. This is additionally worsened by the fact that we cannot synchronize the VPs and the wake up events, as mentioned before. For all these reasons, the slot shifting logic sporadically needs multiple server executions to finish, which explains the observed maximum runtimes.

There are multiple reasons for the high variability of t_{VP}: t_{VP} is not measured in idle slots, because if the current slot is an idle slot, then the routine is not called, since no operations are required. If in the current slot the same job as in the previous slot is scheduled, then our implementation does not need to change the job's scheduling policy or update the VP again (as it has already been done in a previous slot). Instead the logic only sends the continue signal (SIGCONT) to the corresponding job, which results in a much faster runtime. When a job is scheduled the first time, the routine must change its scheduling policy from SCHED_NORMAL to SCHED_EDF, set the jobs budget, period and cpuAffinity parameter[2], and add it to its VP. As a consequence, the observed runtime is much larger in this case, since the RM must use the cgroups interface and potentially requires multiple CBS executions to perform the needed operations.

The table also shows that on average, the handler routine that manages newly incoming aperiodic jobs runs for approximately $3\mu s$. The observed maximum runtime of $23\mu s$, i.e., 0.023% of the slot length, is negligible.

When looking at the start delay t_D of the logic threads, we see that SCHED_EDF Linux was not implemented to satisfy hard real-time requirements in (sub-)millisecond scale: t_D is one order of magnitude larger than t_A. On average Linux schedules the logic threads approximately $272\mu s$ delayed. The observed worst case value of t_D is with $658\mu s$ (0.658% of the slot length) still small. Nevertheless, in future global implementations with smaller slot lengths, $658\mu s$ start delay might be unacceptable.

7.3 Discussion

Our implementation proves that slot shifting can be used to manage scarce resources of multicore platforms. We showed that slot shifting provides deterministic guarantees to applications while allowing for runtime flexibility in a generic adaptive resource management framework. Although SCHED_EDF Linux was not designed with hard real-time guarantees of (sub-)millisecond scale in mind, we demonstrated that even on a 4-years-old standard notebook, slot lengths of 100 milliseconds are possible.

For efficiency reasons, we exchanged the original D-BUS interface between the RM and the applications. The D-BUS interface offers high level services and is thus slow—we observed worst case transaction times in the order of milliseconds. Furthermore, D-BUS makes the RM and the applications depend on a third party: the D-BUS server daemon. D-BUS is not managed by the kernel; instead its daemon is a normal process, i.e., a process without any guarantees on processor time. This leads to two disadvantages: First, unrelated communication of all other applications in the system interferes and slows down the communication with the RM. Second, this interface is subject to resource

[2]The cpuAffinity parameter confines a Linux process to a sub set of cores specified by a bit-mask.

constraints especially under high load conditions of the system, which limits scalability and speed significantly.

For these reasons, our approach features a new interface based on the kill system call that operates much faster. The job starter application and the RM feature a shared memory to transmit aperiodic job parameters to RM. Using these low level system calls, the resulting speed is only limited by the Linux kernel itself and not influenced by any other running processes. Although this interface is not completely free of resource constraints of the sending process, it has proven to run much faster: We observed negligible delays between sending and receiving aperiodic jobs in the order a few microseconds.

As the evaluation showed, using our implementation of slot shifting for ACTORS even smaller slot lengths and less scheduling overheads are theoretically achievable. The efficiency of our implementation is hindered dramatically by the cgroups interface of the RM to SCHED_EDF. Using cgroups is mandatory for the RM to update the VP that hosts the job to be scheduled, but it was not designed with stringent hard real-time requirements in mind. Our measurements showed that adding a process ID to a VP via the cgroups virtual file system in the worst case delays execution by approximately 19.6ms. Essentially, that means that operations on this interface are about one order of magnitude slower than executing the complete slot shifting code. A careful re-design of this interface should allow for interactions below one millisecond. The experiment has shown that then scheduling overheads in the order of one millisecond and thus slot lengths of approximately 10ms could be realized. Using faster and newer state-of-the-art hardware, even shorter slot lengths can be envisioned.

The aim of our approach is to show technical feasibility, thus several restrictions apply: Due to lack of time, the global versions of the slot shifting algorithm have not been implemented. Note that from a technical point of view, there are no constraints imposed by the ACTORS framework that would foreclose their implementation. The runtime experiments in section 5.1 indicate the approximate runtime overhead to be expected due to the global nature of these algorithms. Nevertheless, given the runtime degradation caused by the cgroups interface, this overhead is negligible as it is one order of magnitude below the runtime of the interface.

Another restriction is that our approach does neither allow for feedback from the application to the RM (i.e., the happiness parameter is not considered) nor does our approach monitor the VP's actual resource consumption. Thus, there exists no resource reclaiming mechanism on early completion of jobs within a slot and no dynamic adjustment of reservation or slot length. Furthermore, the scheduler will persistently follow the offline scheduling table, even if a job finishes early and needs less slots than indicated by its WCET.

Another restriction is that the implementation currently only supports a single service level. As future work, a possible extension to support multiple service levels could involve the use of multiple offline scheduling tables. In [91], Theis et al. presented the concept of the *switch-through property* which allows to switch at any moment in time between two different tables.

Chapter 8

Conclusion

Real-time systems are systems that are subject to twofold constraints: First, as every system, they are supposed to show correct output behavior. Second, while they react to stimuli of their environment they must obey pre-defined timing constraints. Both constraints must be satisfied for real-time systems to be considered working functionally correct.

In order to satisfy the timing constraints imposed by the environment, real-time systems employ real-time scheduling algorithms. These algorithms determine when to execute which job of the real-time workload on which processor of the system. There exist many classification schemes for real-time scheduling algorithms. Most importantly, they can be categorized into event- and time-triggered algorithms.

Event-triggered algorithms define a set of rules that are used at runtime of the system to make the scheduling decisions.

Time-triggered algorithms define the execution order of jobs offline, i.e., prior to the runtime of the system. This is often achieved by creation of an offline scheduling table. At runtime, scheduling boils down to dispatching the jobs according to the offline scheduling table.

While event- and time-triggered algorithms provide for all jobs the guarantee that their deadlines will eventually be met, the former category of algorithms does not specify when exactly each specific job is scheduled to execute. Event-triggered algorithms are more flexible to react to unforeseen events such as the arrival of jobs of aperiodic tasks. This flexibility comes at the cost of increased runtime overhead compared to time-triggered systems, which is especially under peak load scenarios undesirable.

Time-triggered algorithms are more predictable, since every action has been pre-planned before the runtime of the system. Another advantage is the temporal isolation among the individual jobs of the tasks, which automatically limits the effects of misbehaving (overrunning) jobs. This is an inherent property of time-triggered systems, as the scheduler recurrently interferes to enforce scheduling decisions. As a result of this predictability and of the temporal isolation, time-triggered systems are easier to certify. The main disadvantage is that, already at design time, the system designer has to specify how the system shall react to all possible future runtime events. This makes the design of time-triggered systems much more demanding than the design of a comparable event-triggered system. Another issue of time-triggered systems is that

they are less flexible to react to unforeseen events such as the arrival of jobs of aperiodic tasks. Finally, adding tasks might require the re-design of the complete offline schedule.

Slot shifting is a real-time scheduling algorithm for distributed systems that combines the benefits of both time- and event-triggered scheduling. Thereby, the algorithm aims at achieving the following goals: First, it resolves the task set's complex constraints, such as, e.g., end-to-end deadlines, by constructing an offline scheduling table. As a result, slot shifting provides predictability and deterministic runtime guarantees for the execution of pre-planned tasks. Second, while ensuring these properties, slot shifting provides flexibility to enable the system to react to unforeseen events. To achieve this runtime behavior, slot shifting aims to guarantee the timely execution of jobs of firm aperiodic tasks. Slot shifting instruments an online acceptance test to determine whether or not there are sufficient resources available to individually guarantee the timely execution of the jobs. This acceptance test itself is based on the information found in the offline scheduling table. The integration of aperiodic jobs into the schedule at runtime must be performed such that the schedulability of already guaranteed jobs is not harmed and such that all jobs' real-time constraints are fulfilled. Finally, while providing all the aforementioned properties, slot shifting aims to minimize the response time of the aperiodic jobs.

In this thesis, we analyzed the original slot shifting algorithm, proposed a faster guarantee algorithm that also improves the responsiveness of aperiodic jobs, and extended the algorithm to handle jobs of non-preemptive firm aperiodic tasks. The main focus of this thesis was set on the design, implementation, and evaluation of global multicore slot shifting algorithms. Furthermore, we proved applicability of the slot shifting algorithm to multicore resource management for soft real-time applications. In the following, we detail the contributions of this thesis. Then, we discuss future work in section 8.2 and conclude the thesis with the final remarks in section 8.3.

8.1 Overview of Contributions

The thesis started with an introduction to real-time scheduling theory and established basic terms and notions. It covered the related work of present real-time scheduling theory which is needed to understand the concepts and approaches presented in this thesis. The following sections summarize our contributions of this thesis to the state-of-the-art in the field of real-time scheduling.

8.1.1 Non-Preemptive Slot Shifting

After describing the original slot shifting algorithm for distributed systems, a first focus of the thesis was to extend this algorithm to handle jobs of firm non-preemptive aperiodic tasks. The most important challenge was to determine whether or not the firm non-preemptive aperiodic job can be guaranteed. In order to do this, a new methodology for a modified acceptance test has been proposed.

The basic idea of our approach is to iterate through the intervals up to the deadline of the non-preemptive job and to determine the maximum available spare capacity

in consecutive slots for non-preemptive execution. Using the set of rules defined in section 3.2.2, our new acceptance test features a runtime complexity of $O(n^2)$.

The main benefits of our approach are that the presented acceptance test has a very low memory overhead and that no new offline scheduling table needs to be calculated.

In principle, there is no need to modify the guarantee algorithm. Nevertheless, when accepting one or multiple non-preemptive jobs, there is a certain risk of a resulting decreased overall acceptance ratio for aperiodic jobs. For this reason, we discussed how to fine-tune the behavior of slot shifting's guarantee algorithm. This enables the system designer to trade flexibility for future aperiodic jobs against improved responsiveness of the non-preemptive jobs. We believe that this trade-off needs to be decided depending on the specific requirements and constraints of the system to be designed.

8.1.2 Multicore Slot Shifting

The main focus of this thesis was set on the design, implementation, and evaluation of multicore slot shifting algorithms. In chapter 4, we introduced and described our approach to global slot shifting algorithms for multicore systems. We examined the challenges that are associated with the design of global slot shifting algorithms. Other researchers have proven the non-existence of an optimal multiprocessor online scheduling algorithm for aperiodic tasks. Hence, our focus was set on finding effective heuristic-based global algorithms with acceptable runtime overheads.

We discussed the issues and bottlenecks of different methods to tackle the following problem: If aperiodic jobs cannot be guaranteed locally, one problem is to quickly determine a suitable core for the aperiodic job. Another challenge is to reduce unnecessary synchronization overheads due to which cores slow down one another. This challenge is closely related to the need to reduce the data exchange among the cores to a minimum. Another issue arises when deciding on how to store the offline scheduling table data. Global storage can be more efficiently implemented, i.e., potentially requires less memory. However, it suffers from delays and overheads due to contention caused by simultaneously accessing cores. Distributed, i.e., core-local storage is usually more resource demanding. On the one hand, it requires more memory and more computational resources, as it forces each core to update its data individually. On the other hand, distributed storage offers the benefit to be less prone to overheads caused by contention.

After discussing the challenges, we finally presented our two global algorithms: a spare-capacity-based and a negotiation-based version of the slot shifting algorithm. Both algorithms employ different heuristic-based approaches to determine the best matching core for an aperiodic job that cannot be guaranteed locally. To avoid contention, both global algorithms store the corresponding offline scheduling table information locally per core. At the same time, they have both been designed such that data exchange is kept at a minimum to avoid overheads.

8.1.3 Evaluation of the Slot Shifting Algorithms on Multicore Systems

One of the main contributions of this thesis consisted of an extensive evaluation of the slot shifting algorithms on multicore systems. The evaluation of the slot shifting algorithms was split into two chapters: In chapter 5, we evaluated the slot shifting algorithms in terms of efficiency. In chapter 6, we focused on quantifying the effectiveness of the algorithms. Both parts together allow for a holistic view of the slot shifting algorithms.

In chapter 5, we analyzed the runtime of the sub functions of the different algorithms using MPARM, a cycle-accurate MP-SoC simulation engine. The aim was to get better insights into the algorithm's runtime behavior. We quantified the cost of feasibly integrating aperiodic jobs at runtime into the schedule. We showed that, as expected, slot shifting's runtime is only minimally affected when the number of offline jobs in the job set is increased. Instead, the number of aperiodic jobs in the job set has a much bigger impact on the observed maximum runtimes of the algorithms. As an additional result of this thorough analysis, we proposed an improved guarantee algorithm (SDL) for slot shifting that yields significant runtime improvements in almost all tested scenarios. Furthermore, we could analyze the runtime overheads of our global algorithms.

In chapter 6, we utilized Elwetritsch, the high performance cluster of the University of Kaiserslautern, to run the experiments with a vast number of randomly created task sets. We run the experiments with the same task sets employing the different slot shifting algorithms and—as a comparison metric—also employing EDF with background processing of the aperiodic jobs. We determined the maximum achievable acceptance ratio within 95% confidence intervals, analyzed the average number of performed acceptance tests and measured the resulting response times of aperiodic jobs. The experiments showed that using SDL significantly improves the response times of the aperiodic jobs for all slot shifting algorithms. This improvement comes at zero cost, since the acceptance ratio remains virtually constant. We not only showed that the partitioned slot shifting always outperforms EDF with background processing of the aperiodic jobs, but also clearly proved the superiority of our global slot shifting algorithms over the original, partitioned algorithm.

8.1.4 Resource Management

The fourth contribution of this thesis was to prove applicability of the slot shifting algorithm to effectively and efficiently perform adaptive resource management in a real-world multicore system. Using a generic adaptive resource management framework, we showed that slot shifting offers a feasible solution to provide deterministic guarantees to soft real-time applications while allowing for runtime flexibility. As a proof of concept, we implemented a slot-shifting-based logic into ACTORS, an adaptive generic resource management framework. We identified the interface between the operating system and the user space to be the crucial bottleneck which limited minimum slot lengths in our implementation. As a result, we confirmed with the measurements performed on our implementation the validity and the technical feasibility of the approach.

8.2 Future Work

The approaches presented in this thesis also show opportunities for future research. One opportunity is the development of global slot shifting algorithms that support jobs of firm non-preemptive aperiodic tasks. Another idea is to drop the need of synchronization of the slots among the cores. Given a future heterogeneous multicore platform, why should cores not be allowed to make progress according to their indivdual speed and computational capabilities? Future algorithms could make use of the hardware's specific cache hierarchy and realize, e.g., a clustered slot shifting approach that makes use of and obeys tasks' cache affinities. Furthermore, future algorithms could extend the notion of "globalness" and guarantee aperiodic jobs by splitting them into sub jobs and migrating them over multiple cores at runtime.

A definitely interesting opportunity is the implementation of slot shifting in a real-time operating system. Another opportunity to explore might be the implementation of slot shifting algorithms as scheduling plugins in *Litmus*^{RT} [92]. Its rich set of tracing tools could give valuable insights in the design of future algorithms.

Also, our slot shifting implementation in the ACTORS framework could be extended in various ways: First of all, there is no technical limitation that prevents from implementing the global algorithms presented in this thesis. A careful re-design of the interface between the resource manager and SCHED_EDF to manage the reservations would allow for much smaller slot sizes and potentially aiming towards providing hard real-time capabilities. Future implementations of a slot shifting logic in ACTORS could allow for multiple service levels, thus offering support for mixed-criticality task sets, e.g., using the concept of the *switch-through property* as presented in [91]. To gain a more complete control over the system, future resource management should also include all the other shared resources of the system, such as caches, core interconnects, memory controllers, hard disks, network cards, etc. Solutions to this might include programmable bus arbiters, locking of cache contents or employment of scratch pads, and the introduction of a notion of fairness at the memory controller level.

8.3 Final Remarks

The advent of multicore platforms unleashes vast computational powers to embedded real-time systems. In order to fully exploit the potential of such massively parallel platforms, future designs require carefully designed real-time scheduling algorithms. Additionally, multicore platforms require real-time operating systems that support resource management and that enforce spatial and temporal isolation among the applications. Resource management helps to avoid costly over-provisioning, to satisfy the needs of the individual applications running concurrently, and to optimize the system's overall QoS. In this thesis, we have designed and implemented two global slot shifting algorithms. We evaluated and proved the capabilities and the potential of our global slot shifting algorithms. By implementing a slot-shifting-based logic into ACTORS, we could close the circle and successfully bridge the gap between real-time scheduling theory and a real-world implementation of a resource management framework.

Appendix A

Detailed Results of MPARM Experiment 4

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category A		
table overhead	—	8304 ±70 (1521)	2542 ±25 (545)
ap job arrival (l)	1940 ±10 (229)	2076 ±12 (263)	2674 ±13 (273)
ap job arrival (d)	—	1742 ±10 (211)	1205 ±5 (110)
update ready list	2378 ±47 (1026)	2092 ±46 (994)	3059 ±42 (899)
select next job	1818 ±12 (259)	2817 ±12 (271)	2765 ±26 (555)
ISR overhead	3189 ±10 (210)	2870 ±11 (235)	2875 ±13 (280)
other	2031	4387	4178
total	11355 ±59 (1300)	24289 ±97 (2112)	19298 ±77 (1649)

Table A.1: *Exp. 4: Overview of measured average execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category B		
table overhead	—	9453 ±222 (1307)	2427 ±97 (562)
ap job arrival (l)	3391 ±63 (376)	4587 ±51 (300)	4456 ±73 (422)
ap job arrival (d)	—	1763 ±34 (203)	1206 ±21 (122)
update ready list	5680 ±255 (1516)	4395 ±233 (1369)	3032 ±144 (837)
acceptance test (l)	9590 ±160 (951)	11642 ±467 (2745)	12465 ±203 (1175)
guarantee alg. (l)	11658 ±949 (5646)	12300 ±1239 (7292)	11640 ±1204 (6978)
select next job	2433 ±43 (256)	3628 ±101 (592)	3296 ±78 (452)
ISR overhead	3394 ±45 (266)	2924 ±36 (213)	2919 ±64 (372)
other	3297	7300	6343
total	39441 ±1212 (7213)	57993 ±1107 (6512)	47783 ±1316 (7628)

Table A.2: *Exp. 4: Overview of measured average execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category C		
table overhead	—	9497 ±450 (1453)	2497 ±108 (550)
ap job arrival (l)	—	2258 ±203 (656)	2854 ±89 (455)
ap job arrival (d)	—	5060 ±104 (336)	4880 ±87 (443)
update ready list	—	4448 ±505 (1631)	3045 ±140 (715)
acceptance test (l)	—	668 ±796 (2568)	807 ±585 (2987)
guarantee alg. (l)	—	700 ±962 (3104)	937 ±742 (3785)
acceptance test (d)	—	12232 ±1172 (3781)	9738 ±541 (2759)
guarantee alg. (d)	—	12370 ±2514 (8113)	4709 ±1391 (7096)
select next job	—	3686 ±173 (557)	3247 ±91 (463)
ISR overhead	—	3233 ±84 (272)	2936 ±78 (398)
other	—	7113	6712
total	—	61264 ±3532 (11398)	42362 ±2388 (12183)

Table A.3: *Exp. 4: Overview of measured average execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category B + C		
table overhead	—	9463 ±199 (1338)	2458 ±72 (557)
ap job arrival (l)	3391 ±63 (376)	4049 ±159 (1066)	3756 ±118 (908)
ap job arrival (d)	—	2526 ±211 (1414)	2810 ±240 (1852)
update ready list	5680 ±255 (1516)	4407 ±213 (1429)	3038 ±102 (784)
acceptance test (l)	9590 ±160 (951)	9105 ±800 (5367)	7374 ±801 (6183)
guarantee alg. (l)	11658 ±949 (5646)	9618 ±1220 (8188)	6966 ±1019 (7865)
acceptance test (d)	—	2828 ±816 (5476)	4252 ±670 (5170)
guarantee alg. (d)	—	2860 ±969 (6503)	2056 ±677 (5229)
select next job	2433 ±43 (256)	3642 ±87 (583)	3275 ±59 (456)
ISR overhead	3394 ±45 (266)	2996 ±39 (262)	2926 ±50 (383)
other	3297	7256	6504
total	39441 ±1212 (7213)	58749 ±1191 (7995)	45416 ±1323 (10217)

Table A.4: *Exp. 4: Overview of measured average execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Overall		
table overhead	—	8404 ±68 (1540)	2532 ±24 (547)
ap job arrival (l)	2038 ±19 (438)	2247 ±30 (685)	2798 ±23 (528)
ap job arrival (d)	—	1810 ±22 (511)	1389 ±36 (814)
update ready list	2602 ±59 (1352)	2293 ±54 (1225)	3056 ±39 (886)
acceptance test (l)	652 ±106 (2427)	788 ±132 (3005)	844 ±138 (3143)
guarantee alg. (l)	793 ±144 (3282)	832 ±159 (3617)	798 ±152 (3461)
acceptance test (d)	—	245 ±79 (1792)	487 ±97 (2210)
guarantee alg. (d)	—	247 ±91 (2070)	235 ±83 (1883)
select next job	1859 ±13 (302)	2889 ±17 (388)	2824 ±25 (568)
ISR overhead	3203 ±10 (221)	2881 ±11 (240)	2881 ±13 (294)
other	2117	4636	4444
total	13265 ±325 (7423)	27270 ±446 (10171)	22288 ±401 (9138)

Table A.5: *Exp. 4: Overview of measured average execution times within 95% confidence intervals, standard deviation in brackets.*

Appendix B

Detailed Results of MPARM Experiment 5

Exec. Time	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category A		
min.	10,705	19,675	15,575
avg.	13,237 ±88 (1,933)	26,751 ±114 (2,484)	19,681 ±90 (1,929)
max.	20,950	37,890	27,115
	Category B		
min.	24,655	47,360	34,450
avg.	42,929 ±1,341 (7,979)	62,581 ±1,137 (6,692)	50,279 ±1,334 (7,671)
max.	54,090	73,695	62,535
	Category C		
min.	–	45,090	29,735
avg.	–	63,193 ±3,476 (11,763)	44,779 ±2,497 (12,865)
max.	–	95,665	72,645
	Overall		
min.	10,705	19,675	15,575
avg.	15,256 ±350 (7,980)	29,936 ±472 (10,773)	22,904 ±431 (9,828)
max.	54,090	95,665	72,645

Table B.1: *Experiment 5a: 70 offline jobs, 30 aperiodic jobs. Overview of measured execution times (in ns) within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category A		
table overhead	—	10888 ±78 (1692)	2535 ±24 (518)
ap job arrival (l)	2210 ±10 (218)	2034 ±17 (362)	2606 ±13 (286)
ap job arrival (d)	—	1501 ±5 (112)	1206 ±5 (107)
update ready list	2930 ±72 (1578)	2285 ±66 (1441)	3489 ±55 (1190)
select next job	2300 ±16 (355)	2888 ±16 (353)	2696 ±30 (637)
ISR overhead	3727 ±12 (273)	2745 ±10 (224)	2826 ±13 (272)
other	2069.4	4410.3	4321.6
total	13237 ±88 (1933)	26751 ±114 (2484)	19681 ±90 (1929)

Table B.2: *Experiment 5a: 70 offline jobs, 30 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category B		
table overhead	—	11817 ±217 (1276)	2309 ±73 (419)
ap job arrival (l)	4416 ±53 (318)	4728 ±76 (450)	5054 ±62 (358)
ap job arrival (d)	—	1491 ±18 (108)	1203 ±15 (88)
update ready list	6192 ±346 (2059)	5085 ±316 (1861)	3484 ±216 (1242)
acceptance test (l)	9956 ±168 (998)	11885 ±508 (2990)	12858 ±205 (1179)
guarantee alg. (l)	12449 ±1061 (6313)	13685 ±1271 (7479)	12543 ±1275 (7330)
select next job	2666 ±54 (321)	3785 ±107 (629)	3364 ±95 (545)
ISR overhead	3910 ±54 (322)	3014 ±42 (247)	3043 ±56 (321)
other	3339.6	7091.3	6422.5
total	42929 ±1341 (7979)	62581 ±1137 (6692)	50279 ±1334 (7671)

Table B.3: *Experiment 5a: 70 offline jobs, 30 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category C		
table overhead	—	11758 ±480 (1624)	2529 ±92 (475)
ap job arrival (l)	—	2264 ±206 (697)	2886 ±148 (760)
ap job arrival (d)	—	4818 ±159 (537)	4979 ±158 (812)
update ready list	—	4762 ±609 (2060)	3561 ±209 (1076)
acceptance test (l)	—	1038 ±1149 (3887)	1091 ±686 (3537)
guarantee alg. (l)	—	—	683 ±662 (3412)
acceptance test (d)	—	14044 ±1850 (6263)	10724 ±755 (3888)
guarantee alg. (d)	—	10829 ±2639 (8931)	5306 ±1543 (7950)
select next job	—	3658 ±194 (658)	3206 ±118 (608)
ISR overhead	—	3045 ±69 (234)	2912 ±70 (359)
other	—	6976.9	6900.8
total	—	63193 ±3476 (11763)	44779 ±2497 (12865)

Table B.4: *Experiment 5a: 70 offline jobs, 30 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category B + C		
table overhead	—	11802 ±201 (1366)	2407 ±59 (457)
ap job arrival (l)	4416 ±53 (318)	4115 ±175 (1188)	4088 ±158 (1222)
ap job arrival (d)	—	2318 ±216 (1469)	2885 ±254 (1958)
update ready list	6192 ±346 (2059)	5005 ±282 (1911)	3518 ±151 (1169)
acceptance test (l)	9956 ±168 (998)	9189 ±840 (5701)	7617 ±826 (6377)
guarantee alg. (l)	12449 ±1061 (6313)	10283 ±1294 (8782)	7260 ±1082 (8351)
acceptance test (d)	—	3491 ±1006 (6829)	4777 ±769 (5935)
guarantee alg. (d)	—	2692 ±949 (6443)	2363 ±766 (5914)
select next job	2666 ±54 (321)	3753 ±94 (637)	3293 ±75 (578)
ISR overhead	3910 ±54 (322)	3022 ±36 (243)	2985 ±45 (344)
other	3339.6	7062.9	6635.5
total	42929 ±1341 (7979)	62733 ±1210 (8214)	47829 ±1379 (10646)

Table B.5: *Experiment 5a: 70 offline jobs, 30 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Overall		
table overhead	—	10969 ±74 (1686)	2521 ±22 (513)
ap job arrival (l)	2360 ±26 (600)	2218 ±34 (770)	2776 ±30 (682)
ap job arrival (d)	—	1573 ±22 (505)	1399 ±38 (856)
update ready list	3152 ±79 (1811)	2526 ±73 (1677)	3492 ±52 (1188)
acceptance test (l)	677 ±110 (2520)	813 ±136 (3111)	872 ±142 (3244)
guarantee alg. (l)	847 ±155 (3538)	910 ±172 (3915)	831 ±160 (3647)
acceptance test (d)	—	309 ±99 (2256)	547 ±110 (2516)
guarantee alg. (d)	—	238 ±90 (2059)	271 ±94 (2135)
select next job	2325 ±16 (365)	2965 ±20 (458)	2764 ±29 (659)
ISR overhead	3740 ±12 (281)	2770 ±10 (239)	2845 ±13 (285)
other	2155.8	4645.1	4586.5
total	15256 ±350 (7980)	29936 ±472 (10773)	22904 ±431 (9828)

Table B.6: *Experiment 5a: 70 offline jobs, 30 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Exec. Time	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category A		
min.	11,085	20,610	15,560
avg.	13,440 ±73 (1,514)	25,167 ±120 (2,485)	19,686 ±88 (1,795)
max.	22,540	35,400	30,195
	Category B		
min.	24,365	42,490	32,425
avg.	42,679 ±499 (4,643)	59,687 ±565 (5,208)	49,796 ±544 (4,977)
max.	53,775	72,395	62,110
	Category C		
min.	–	44,345	28,885
avg.	–	63,827 ±4,213 (12,715)	43,350 ±2,653 (12,696)
max.	–	96,595	88,305
	Overall		
min.	11,085	20,610	15,560
avg.	18,293 ±488 (11,131)	31,470 ±609 (13,892)	25,560 ±535 (12,217)
max.	53,775	96,595	88,305

Table B.7: *Experiment 5b: 30 offline jobs, 80 aperiodic jobs. Overview of measured execution times (in ns) within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category B		
table overhead	—	9963 ±140 (1287)	2347 ±38 (351)
ap job arrival (l)	5177 ±40 (374)	5184 ±43 (393)	5581 ±35 (322)
ap job arrival (d)	—	1342 ±24 (218)	1277 ±13 (120)
update ready list	6494 ±155 (1439)	6092 ±154 (1421)	3271 ±110 (1007)
acceptance test (l)	8929 ±89 (825)	9643 ±212 (1954)	11675 ±113 (1031)
guarantee alg. (l)	11723 ±350 (3255)	13203 ±488 (4493)	12672 ±450 (4114)
select next job	2498 ±24 (221)	3511 ±49 (449)	2966 ±43 (393)
ISR overhead	4061 ±25 (236)	3356 ±46 (428)	3445 ±38 (343)
other	3797.4	7393.9	6561.6
total	42679 ±499 (4643)	59687 ±565 (5208)	49796 ±544 (4977)

Table B.9: *Experiment 5b: 30 offline jobs, 80 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category C		
table overhead	—	9821 ±321 (968)	2504 ±97 (464)
ap job arrival (l)	—	2651 ±404 (1219)	2984 ±209 (999)
ap job arrival (d)	—	4717 ±80 (240)	4653 ±103 (494)
update ready list	—	5798 ±512 (1546)	3381 ±275 (1315)
acceptance test (l)	—	1665 ±1294 (3906)	1388 ±774 (3704)
guarantee alg. (l)	—	1711 ±1610 (4861)	1254 ±838 (4009)
acceptance test (d)	—	10737 ±1053 (3179)	10084 ±675 (3230)
guarantee alg. (d)	—	12387 ±1940 (5855)	4468 ±1351 (6467)
select next job	—	3462 ±143 (433)	2773 ±105 (504)
ISR overhead	—	3218 ±107 (322)	3153 ±60 (289)
other	0.0	7661.1	6708.0
total	—	63827 ±4213 (12715)	43350 ±2653 (12696)

Table B.10: *Experiment 5b: 30 offline jobs, 80 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category A		
table overhead	—	9135 ±70 (1452)	2492 ±20 (409)
ap job arrival (l)	2228 ±18 (382)	2101 ±20 (423)	2627 ±18 (358)
ap job arrival (d)	—	1363 ±11 (235)	1287 ±6 (128)
update ready list	2701 ±56 (1163)	2116 ±49 (1017)	3201 ±50 (1009)
select next job	2269 ±12 (250)	2809 ±14 (297)	2525 ±22 (452)
ISR overhead	3600 ±14 (290)	2951 ±16 (339)	2967 ±16 (319)
other	2642.2	4691.7	4588.2
total	13439 ±73 (1514)	25167 ±120 (2485)	19686 ±88 (1795)

Table B.8: *Experiment 5b: 30 offline jobs, 80 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Category B + C		
table overhead	—	9949 ±130 (1259)	2381 ±37 (383)
ap job arrival (l)	5177 ±40 (374)	4938 ±95 (918)	5022 ±116 (1198)
ap job arrival (d)	—	1669 ±106 (1024)	2003 ±137 (1412)
update ready list	6494 ±155 (1439)	6063 ±148 (1434)	3295 ±105 (1080)
acceptance test (l)	8929 ±89 (825)	8869 ±334 (3237)	9462 ±451 (4655)
guarantee alg. (l)	11723 ±350 (3255)	12089 ±584 (5661)	10215 ±603 (6227)
acceptance test (d)	—	1041 ±343 (3328)	2170 ±427 (4409)
guarantee alg. (d)	—	1201 ±422 (4088)	961 ±340 (3507)
select next job	2498 ±24 (221)	3506 ±46 (447)	2925 ±41 (426)
ISR overhead	4061 ±25 (236)	3343 ±43 (420)	3382 ±34 (353)
other	3797.4	7419.8	6593.1
total	42679 ±499 (4643)	60089 ±663 (6424)	48409 ±756 (7800)

Table B.11: *Experiment 5b: 30 offline jobs, 80 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

Sub Function	Orig. Part. Slot Shifting	Global Algorithm 1	Global Algorithm 2
	Overall		
table overhead	—	9282 ±64 (1453)	2469 ±18 (407)
ap job arrival (l)	2718 ±51 (1162)	2613 ±53 (1221)	3116 ±51 (1153)
ap job arrival (d)	—	1418 ±22 (498)	1433 ±31 (709)
update ready list	3330 ±82 (1862)	2829 ±82 (1877)	3220 ±45 (1024)
acceptance test (l)	1482 ±146 (3340)	1601 ±161 (3678)	1935 ±191 (4358)
guarantee alg. (l)	1946 ±200 (4559)	2182 ±229 (5234)	2089 ±219 (4990)
acceptance test (d)	—	188 ±64 (1468)	444 ±95 (2176)
guarantee alg. (d)	—	217 ±79 (1795)	197 ±71 (1631)
select next job	2307 ±11 (259)	2935 ±19 (425)	2606 ±21 (475)
ISR overhead	3676 ±14 (330)	3021 ±17 (386)	3052 ±16 (367)
other	2834.0	5184.1	4998.2
total	18293 ±488 (11131)	31470 ±609 (13892)	25560 ±535 (12217)

Table B.12: *Experiment 5b: 30 offline jobs, 80 aperiodic jobs. Overview of measured execution times within 95% confidence intervals, standard deviation in brackets.*

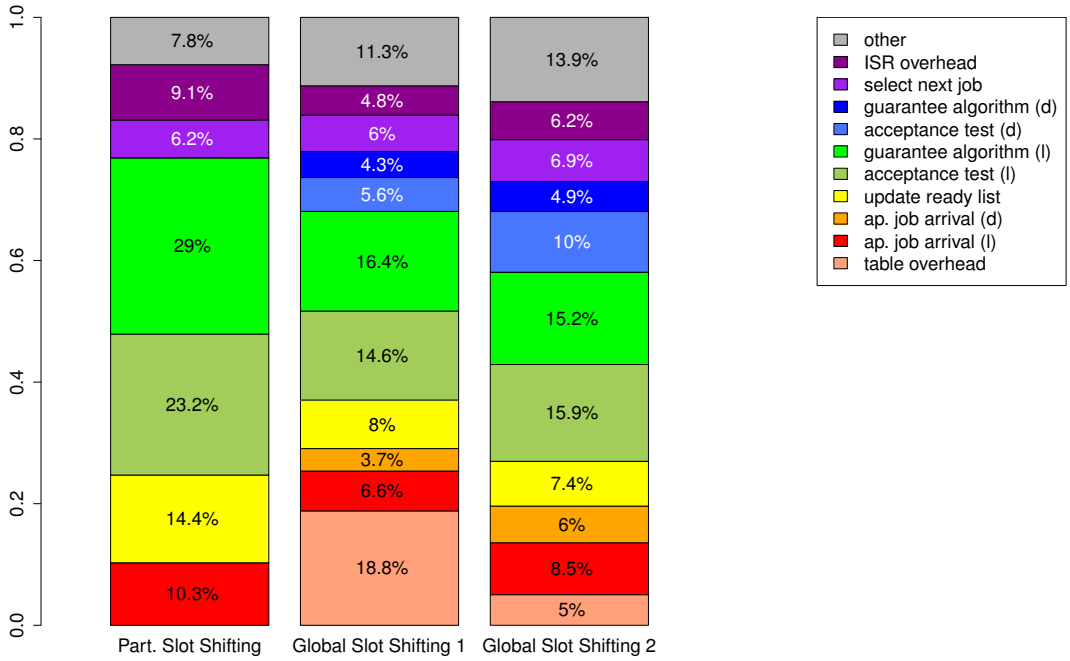


Figure B.1: *Exp. 5a: Normalized runtimes for partitioned and global slot shifting for slots of category B and C, job set with 70 instead of 30 offline jobs per core.*

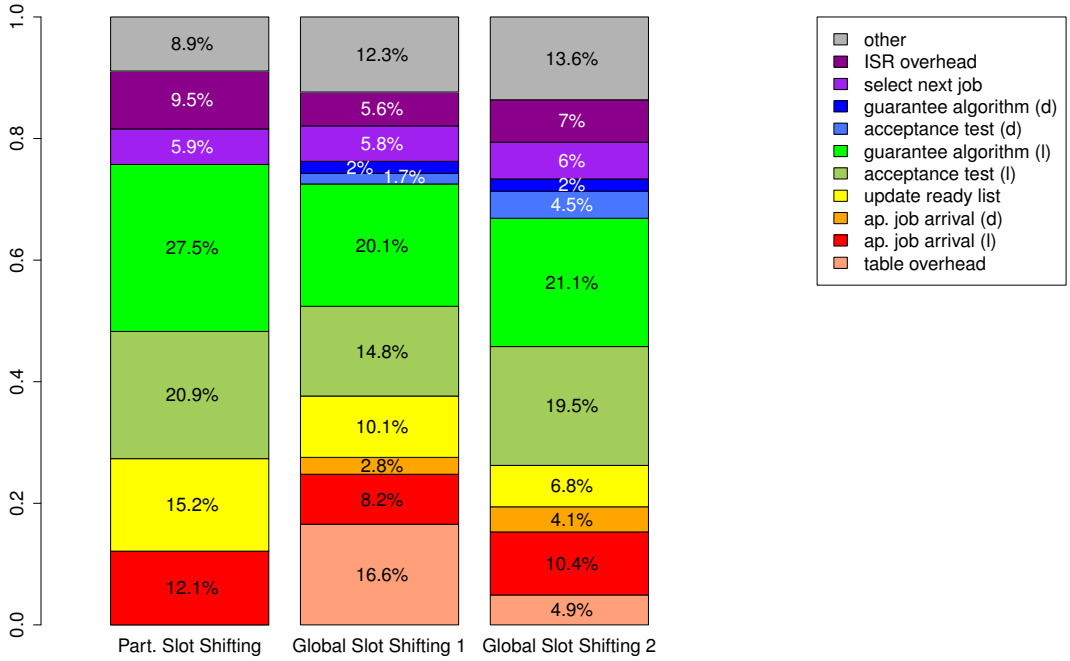


Figure B.2: *Exp. 5b: Normalized runtimes for partitioned and global slot shifting for slots of category B and C, job set with 80 instead of 30 aperiodic jobs per core.*

Appendix C

Detailed Results of Linux Experiment 1

This appendix lists the results of the effectiveness measurements we performed on Elwetritsch, the high performance cluster of the University of Kaiserslautern:
Experiment 1 - This experiment runs on a simulated 4-core system, the offline utilization on all cores is balanced, i.e., the same for all cores. Table C.1 lists the parameters of the task set.

All the other tables and figures list the results for three distinct scenarios which differ in the utilization created by the arriving aperiodic jobs: 10%, 10%, and 50%. The experiment has been conducted for offline utilizations between 0% and 90% in steps of 10%; each result being based on 1000 simulated jobs sets. Furthermore, each table lists the results for DLX factors of 1, 1.5, 2, and 5. The following scheduling algorithms have been applied:

Parameter	Offline Tasks	Aperiodic Tasks
WCET	$1 - 15$	$10 - 15$
Period/Deadline	$15 - 30$	$[1; 1.5; 2; 5] * WCET$
Resulting U per core	0%, 10%, 10%, ... 90%	10%, 10%, 50%

Table C.1: *Linux Experiment 1: overview of the task set parameters.*

- EDF with background processing of aperiodic jobs
- partitioned slot shifting
- global spare-capacity-based slot shifting (global algorithm 1)
- global negotiation-based slot shifting (global algorithm 2).

C.1 Acceptance Ratio

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 1	EDF	92.27	46.85	23.24	13.43	7.85	3.58	0.71	0.23	0.10	0.07
DLX 1	part.	92.61	92.16	90.97	88.46	81.94	69.61	51.86	35.43	19.15	5.65
	global 1	99.90	99.88	99.82	99.68	98.75	94.82	82.74	63.70	38.37	12.67
	global 2	99.97	99.96	99.92	99.81	99.02	95.68	84.25	65.69	39.80	13.00
Exp. 1	EDF	95.57	93.65	90.12	62.75	26.38	9.43	1.68	0.37	0.09	0.05
DLX 1.5	part.	95.81	95.52	94.74	93.58	91.21	82.77	67.42	48.89	26.82	8.19
	global 1	99.99	100	99.99	99.98	99.93	99.35	94.83	80.25	52.23	18.30
	global 2	99.99	100	99.99	99.98	99.93	99.39	95.09	80.84	52.73	18.42
Exp. 1	EDF	99.67	98.55	95.68	92.33	85.17	51.92	11.81	0.85	0.11	0.04
DLX 2	part.	99.68	99.51	98.97	97.75	95.82	92.12	84.94	67.75	38.12	10.73
	global 1	100	100	100	100	99.99	99.95	99.49	94.24	66.66	22.83
	global 2	100	100	100	100	99.99	99.95	99.51	94.50	67.03	23.02
Exp. 1	EDF	100	100	100	99.99	99.93	99.52	96.70	82.55	38.47	0.19
DLX 5	part.	100	100	100	100	99.98	99.91	99.21	95.41	80.32	35.17
	global 1	100	100	100	100	100	100	100	99.98	98.09	55.81
	global 2	100	100	100	100	100	100	100	99.98	97.98	55.88

Table C.2: *Linux Experiment 1: measured acceptance ratio (in %),* $U_{aperiodic} = 10\%$.

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 1	EDF	84.57	42.70	21.11	12.45	7.30	3.24	0.72	0.19	0.11	0.07
DLX 1	part.	85.86	85.16	83.60	80.54	74.16	62.82	47.21	32.38	16.73	5.10
	global 1	99.29	99.14	98.86	98.20	96.02	89.40	74.73	54.84	30.79	10.04
	global 2	99.54	99.44	99.22	98.64	96.76	90.63	76.37	56.54	31.82	10.23
Exp. 1	EDF	90.53	86.60	81.28	55.28	23.16	8.50	1.58	0.34	0.12	0.05
DLX 1.5	part.	91.61	90.76	89.43	87.14	83.27	74.55	60.35	42.46	23.52	6.80
	global 1	99.94	99.90	99.86	99.66	99.16	96.75	88.21	68.82	41.71	13.57
	global 2	99.94	99.90	99.85	99.66	99.12	96.79	88.52	69.31	42.01	13.63
Exp. 1	EDF	98.48	95.71	89.94	84.06	74.22	43.83	9.84	0.72	0.09	0.05
DLX 2	part.	98.65	97.97	96.70	94.26	90.27	84.41	74.75	57.72	31.62	9.06
	global 1	100	100	100	99.98	99.87	99.25	96.08	83.28	51.35	17.10
	global 2	100	100	100	99.98	99.85	99.20	96.04	83.49	51.58	17.21
Exp. 1	EDF	100	100	99.99	99.87	99.31	96.42	87.30	63.62	24.69	0.13
DLX 5	part.	100	100	99.99	99.96	99.79	98.91	95.36	85.28	63.43	25.70
	global 1	100	100	100	100	100	100	99.96	98.41	81.52	36.41
	global 2	100	100	100	100	100	100	99.96	98.26	81.21	36.50

Table C.3: *Linux Experiment 1: measured acceptance ratio (in %),* $U_{aperiodic} = 20\%$.

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 1	EDF	64.34	32.00	15.90	9.38	5.52	2.47	0.54	0.17	0.08	0.05
DLX 1	part.	70.30	69.25	67.18	63.49	57.46	47.77	35.65	23.48	12.56	3.80
	global 1	92.01	91.06	88.96	85.02	78.58	67.85	52.64	35.48	19.87	6.36
	global 2	92.74	91.79	89.69	85.92	79.63	69.08	53.82	36.43	20.37	6.45
Exp. 1	EDF	73.64	66.43	58.55	38.24	16.51	6.22	1.14	0.25	0.07	0.04
DLX 1.5	part.	79.63	77.98	75.23	71.14	65.51	56.61	44.09	30.57	16.36	4.94
	global 1	97.70	96.95	95.36	92.26	86.76	77.28	62.48	44.42	25.16	8.39
	global 2	97.63	96.82	95.12	91.92	86.43	77.11	62.56	44.66	25.26	8.42
Exp. 1	EDF	88.82	81.28	69.97	59.64	48.20	26.59	5.89	0.49	0.07	0.03
DLX 2	part.	91.51	89.20	85.68	80.22	73.33	64.49	53.36	38.95	20.97	6.26
	global 1	99.75	99.43	98.70	96.47	91.84	82.91	69.57	51.72	29.18	10.02
	global 2	99.75	99.39	98.60	96.24	91.42	82.52	69.32	51.73	29.23	10.06
Exp. 1	EDF	99.84	99.30	97.41	92.72	82.57	66.05	45.26	24.15	6.77	0.05
DLX 5	part.	99.87	99.59	98.66	96.39	91.21	81.87	68.74	52.98	35.44	14.36
	global 1	100	100	100	99.94	98.78	90.99	75.21	56.74	38.60	17.53
	global 2	100	100	100	99.93	98.63	90.68	75.05	56.68	38.57	17.53

Table C.4: *Linux Experiment 1: measured acceptance ratio (in %), $U_{aperiodic}$ = 50%.*

Offline Utilization	EDF with Backgr. Proc.	Partitioned Slot Shifting	Global Slot Shifting 1	Global Slot Shifting 2
0%	99.670 ± 0.043	99.684 ± 0.041	100.000 ± 0.000	100.000 ± 0.000
10%	98.550 ± 0.097	99.506 ± 0.055	100.000 ± 0.000	100.000 ± 0.000
20%	95.679 ± 0.153	98.967 ± 0.075	100.000 ± 0.000	100.000 ± 0.000
30%	92.325 ± 0.205	97.752 ± 0.114	99.998 ± 0.004	99.998 ± 0.004
40%	85.168 ± 0.282	95.823 ± 0.153	99.993 ± 0.007	99.993 ± 0.007
50%	51.924 ± 0.380	92.123 ± 0.203	99.951 ± 0.016	99.947 ± 0.017
60%	11.808 ± 0.275	84.939 ± 0.285	99.493 ± 0.065	99.511 ± 0.063
70%	0.850 ± 0.080	67.749 ± 0.592	94.241 ± 0.331	94.497 ± 0.318
80%	0.112 ± 0.026	38.122 ± 0.727	66.662 ± 0.866	67.032 ± 0.861
90%	0.045 ± 0.015	10.733 ± 0.376	22.831 ± 0.592	23.023 ± 0.601

Table C.5: *Linux Experiment 1: acceptance ratios with 95% confidence interval ($U_{aperiodic}$ = 10%, DLX factor 2).*

Offline Utilization	EDF with Backgr. Proc.	Partitioned Slot Shifting	Global Slot Shifting 1	Global Slot Shifting 2
0%	98.484 ± 0.073	98.645 ± 0.062	99.999 ± 0.001	100.000 ± 0.001
10%	95.714 ± 0.118	97.965 ± 0.078	99.999 ± 0.001	99.999 ± 0.001
20%	89.942 ± 0.176	96.698 ± 0.098	99.998 ± 0.002	99.998 ± 0.002
30%	84.058 ± 0.209	94.256 ± 0.127	99.980 ± 0.007	99.975 ± 0.008
40%	74.224 ± 0.238	90.267 ± 0.157	99.871 ± 0.022	99.853 ± 0.025
50%	43.827 ± 0.265	84.412 ± 0.176	99.250 ± 0.053	99.199 ± 0.054
60%	9.839 ± 0.171	74.754 ± 0.254	96.076 ± 0.131	96.044 ± 0.129
70%	0.718 ± 0.058	57.724 ± 0.463	83.279 ± 0.419	83.494 ± 0.405
80%	0.092 ± 0.017	31.616 ± 0.561	51.345 ± 0.699	51.578 ± 0.697
90%	0.054 ± 0.015	9.057 ± 0.298	17.095 ± 0.422	17.213 ± 0.425

Table C.6: *Linux Experiment 1: acceptance ratios with 95% confidence interval ($U_{aperiodic}$ = 20%, DLX factor 2).*

Offline Utilization	EDF with Backgr. Proc.	Partitioned Slot Shifting	Global Slot Shifting 1	Global Slot Shifting 2
0%	88.823 ± 0.127	91.510 ± 0.088	99.754 ± 0.022	99.746 ± 0.022
10%	81.279 ± 0.155	89.196 ± 0.097	99.431 ± 0.036	99.393 ± 0.037
20%	69.968 ± 0.163	85.678 ± 0.101	98.697 ± 0.052	98.597 ± 0.054
30%	59.636 ± 0.148	80.221 ± 0.111	96.473 ± 0.075	96.245 ± 0.075
40%	48.198 ± 0.150	73.333 ± 0.121	91.836 ± 0.102	91.423 ± 0.103
50%	26.587 ± 0.153	64.491 ± 0.129	82.913 ± 0.115	82.518 ± 0.116
60%	5.885 ± 0.098	53.363 ± 0.151	69.565 ± 0.124	69.321 ± 0.122
70%	0.494 ± 0.031	38.952 ± 0.259	51.717 ± 0.238	51.731 ± 0.230
80%	0.072 ± 0.011	20.974 ± 0.317	29.181 ± 0.370	29.231 ± 0.370
90%	0.028 ± 0.006	6.260 ± 0.172	10.024 ± 0.219	10.064 ± 0.219

Table C.7: *Linux Experiment 1: acceptance ratios with 95% confidence interval ($U_{aperiodic}$ = 50%, DLX factor 2).*

C.2 Number of Acceptance Tests

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.08	1.09	1.10	1.13	1.22	1.39	1.70	2.06	2.46	2.83
	glob. alg. 2	1.11	1.11	1.14	1.19	1.31	1.62	2.16	2.68	3.30	3.79
DLX 1.5	glob. alg. 1	1.04	1.05	1.06	1.07	1.10	1.21	1.49	2.00	2.76	3.56
	glob. alg. 2	1.06	1.06	1.07	1.10	1.13	1.28	1.70	2.28	3.02	3.70
DLX 2	glob. alg. 1	1.00	1.00	1.01	1.02	1.04	1.09	1.19	1.52	2.40	3.46
	glob. alg. 2	1.00	1.01	1.01	1.03	1.06	1.12	1.26	1.73	2.75	3.62
DLX 5	glob. alg. 1	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.30	2.68
	glob. alg. 2	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.07	1.43	2.97

Table C.8: *Linux Experiment 1: average number of acceptance tests per aperiodic job, $U_{aperiodic}$ = 10%.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	glob. alg. 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	glob. alg. 1	4.00	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.98	3.97
	glob. alg. 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	glob. alg. 1	–	–	–	4.00	4.00	3.98	4.00	3.99	3.98	3.97
	glob. alg. 2	–	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	glob. alg. 1	–	–	–	–	–	–	–	4.00	3.98	3.96
	glob. alg. 2	–	–	–	–	–	–	–	4.00	4.00	4.00

Table C.9: *Linux Experiment 1: average number of acceptance tests per finally rejected aperiodic job, $U_{aperiodic}$ = 10%.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.08	1.08	1.10	1.13	1.19	1.31	1.44	1.53	1.61	1.67
	glob. alg. 2	1.11	1.11	1.14	1.18	1.28	1.52	1.84	2.02	2.26	2.40
DLX 1.5	glob. alg. 1	1.04	1.05	1.06	1.07	1.10	1.19	1.36	1.51	1.66	1.71
	glob. alg. 2	1.06	1.06	1.07	1.10	1.13	1.27	1.59	1.89	2.16	2.41
DLX 2	glob. alg. 1	1.00	1.00	1.01	1.02	1.04	1.09	1.17	1.37	1.62	1.72
	glob. alg. 2	1.00	1.01	1.01	1.03	1.06	1.12	1.25	1.61	2.16	2.37
DLX 5	glob. alg. 1	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.24	1.66
	glob. alg. 2	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.07	1.38	2.17

Table C.10: *Linux Experiment 1: average number of acceptance tests per accepted aperiodic job, $U_{aperiodic} = 10\%$.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.18	1.18	1.21	1.26	1.36	1.56	1.87	2.21	2.57	2.87
	glob. alg. 2	1.24	1.26	1.31	1.37	1.55	1.91	2.41	2.93	3.44	3.84
DLX 1.5	glob. alg. 1	1.10	1.11	1.13	1.16	1.22	1.39	1.75	2.33	3.01	3.65
	glob. alg. 2	1.12	1.14	1.18	1.23	1.32	1.56	1.99	2.62	3.26	3.78
DLX 2	glob. alg. 1	1.01	1.02	1.03	1.06	1.11	1.21	1.42	1.91	2.78	3.57
	glob. alg. 2	1.02	1.03	1.04	1.09	1.16	1.30	1.60	2.20	3.05	3.72
DLX 5	glob. alg. 1	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.24	1.95	3.14
	glob. alg. 2	1.00	1.00	1.00	1.00	1.00	1.01	1.07	1.36	2.26	3.37

Table C.11: *Linux Experiment 1: average number of acceptance tests per aperiodic job, $U_{aperiodic} = 20\%$.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	glob. alg. 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	glob. alg. 1	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.96	3.95
	glob. alg. 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	glob. alg. 1	4.00	4.00	4.00	4.00	3.99	3.99	3.99	3.98	3.97	3.95
	glob. alg. 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	glob. alg. 1	–	–	–	–	–	–	4.00	3.99	3.98	3.95
	glob. alg. 2	–	–	–	–	–	–	4.00	4.00	4.00	4.00

Table C.12: *Linux Experiment 1: average number of acceptance tests per finally rejected aperiodic job, $U_{aperiodic} = 20\%$.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.16	1.17	1.19	1.22	1.29	1.39	1.49	1.56	1.63	1.68
	glob. alg. 2	1.22	1.24	1.29	1.34	1.47	1.70	1.93	2.13	2.28	2.44
DLX 1.5	glob. alg. 1	1.09	1.10	1.12	1.15	1.20	1.31	1.45	1.59	1.68	1.72
	glob. alg. 2	1.12	1.14	1.17	1.22	1.30	1.48	1.74	2.02	2.25	2.40
DLX 2	glob. alg. 1	1.01	1.02	1.03	1.06	1.11	1.19	1.32	1.50	1.66	1.73
	glob. alg. 2	1.02	1.03	1.04	1.09	1.15	1.28	1.50	1.84	2.17	2.41
DLX 5	glob. alg. 1	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.19	1.49	1.72
	glob. alg. 2	1.00	1.00	1.00	1.00	1.00	1.01	1.07	1.31	1.85	2.30

Table C.13: *Linux Experiment 1: average number of acceptance tests per accepted aperiodic job, $U_{aperiodic}$ = 20%.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.48	1.51	1.56	1.66	1.79	2.00	2.24	2.50	2.73	2.91
	glob. alg. 2	1.79	1.83	1.89	2.10	2.31	2.64	2.98	3.35	3.66	3.90
DLX 1.5	glob. alg. 1	1.32	1.36	1.45	1.58	1.78	2.09	2.50	2.94	3.38	3.73
	glob. alg. 2	1.46	1.51	1.61	1.79	2.06	2.41	2.82	3.21	3.58	3.87
DLX 2	glob. alg. 1	1.10	1.15	1.22	1.36	1.58	1.91	2.31	2.77	3.30	3.70
	glob. alg. 2	1.16	1.20	1.33	1.50	1.80	2.19	2.63	3.07	3.52	3.84
DLX 5	glob. alg. 1	1.00	1.00	1.01	1.04	1.16	1.54	2.10	2.64	3.09	3.55
	glob. alg. 2	1.00	1.01	1.02	1.06	1.25	1.75	2.40	2.90	3.34	3.72

Table C.14: *Linux Experiment 1: average number of acceptance tests per aperiodic job, $U_{aperiodic}$ = 50%..*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	glob. alg. 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	glob. alg. 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.94	3.92
	glob. alg. 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	glob. alg. 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.96	3.95	3.92
	glob. alg. 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	glob. alg. 1	–	–	4.00	4.00	4.00	3.99	3.99	3.98	3.97	3.93
	glob. alg. 2	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Table C.15: *Linux Experiment 1: average number of acceptance tests per finally rejected aperiodic job, $U_{aperiodic}$ = 50%.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.35	1.36	1.38	1.42	1.46	1.52	1.57	1.60	1.64	1.68
	glob. alg. 2	1.62	1.64	1.65	1.79	1.88	2.04	2.10	2.23	2.34	2.41
DLX 1.5	glob. alg. 1	1.26	1.28	1.32	1.38	1.45	1.54	1.62	1.68	1.72	1.73
	glob. alg. 2	1.40	1.42	1.49	1.60	1.75	1.94	2.11	2.23	2.35	2.42
DLX 2	glob. alg. 1	1.10	1.13	1.18	1.26	1.37	1.48	1.59	1.67	1.73	1.74
	glob. alg. 2	1.15	1.19	1.29	1.40	1.60	1.81	2.03	2.20	2.36	2.45
DLX 5	glob. alg. 1	1.00	1.00	1.01	1.04	1.12	1.29	1.48	1.61	1.70	1.77
	glob. alg. 2	1.00	1.01	1.02	1.06	1.21	1.52	1.87	2.07	2.30	2.39

Table C.16: *Linux Experiment 1: average number of acceptance tests per accepted aperiodic job,* $U_{aperiodic} = 50\%$.

C.3 Quickness

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	EDF	0.99	0.53	0.69	0.76	0.75	0.71	0.67	0.71	0.67	0.82
	partitioned	0.99	0.99	0.99	0.98	0.98	0.96	0.93	0.91	0.90	0.91
	global 1	0.91	0.91	0.89	0.86	0.79	0.66	0.53	0.43	0.36	0.29
	global 2	0.91	0.90	0.89	0.86	0.78	0.66	0.51	0.41	0.34	0.28
DLX 1.5	EDF	0.97	0.76	0.51	0.38	0.44	0.51	0.49	0.48	0.50	0.44
	partitioned	0.97	0.96	0.95	0.92	0.88	0.83	0.77	0.73	0.72	0.73
	global 1	0.97	0.96	0.94	0.91	0.86	0.80	0.72	0.66	0.62	0.62
	global 2	0.96	0.95	0.93	0.91	0.86	0.80	0.72	0.66	0.62	0.63
DLX 2	EDF	0.96	0.85	0.73	0.58	0.40	0.26	0.18	0.33	0.27	0.22
	partitioned	0.96	0.93	0.89	0.83	0.77	0.69	0.60	0.53	0.51	0.55
	global 1	0.96	0.93	0.89	0.83	0.77	0.69	0.59	0.52	0.47	0.51
	global 2	0.96	0.92	0.89	0.83	0.77	0.68	0.58	0.51	0.47	0.50
DLX 5	EDF	0.99	0.96	0.92	0.87	0.81	0.71	0.56	0.40	0.16	0.13
	partitioned	0.99	0.96	0.92	0.87	0.81	0.71	0.56	0.39	0.23	0.16
	global 1	0.99	0.96	0.92	0.87	0.81	0.71	0.56	0.39	0.23	0.15
	global 2	0.99	0.96	0.92	0.87	0.81	0.71	0.56	0.39	0.22	0.14

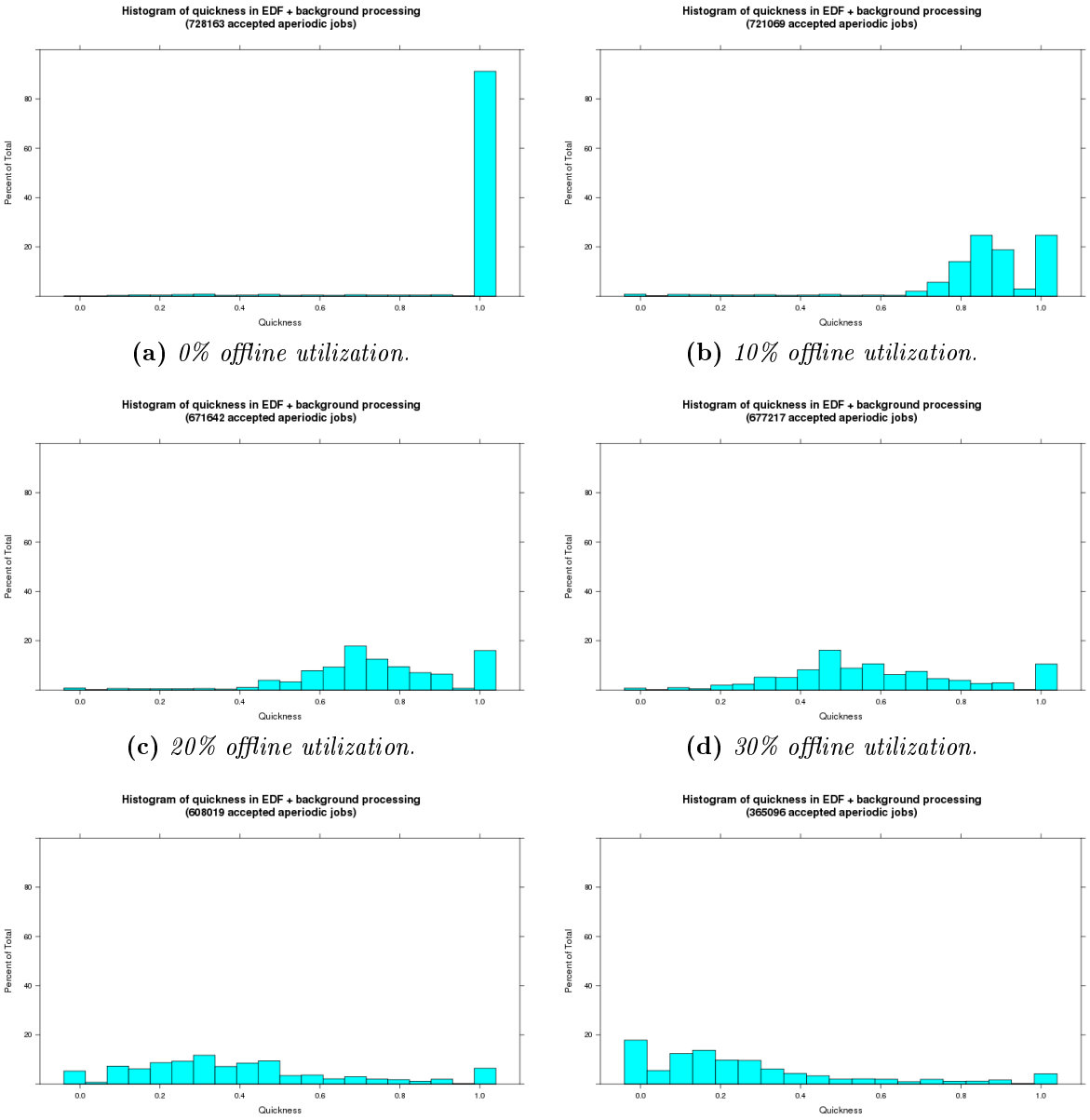
Table C.17: *Linux Experiment 1: average quickness,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	EDF	0.98	0.54	0.70	0.76	0.75	0.70	0.64	0.73	0.78	0.75
	partitioned	0.98	0.98	0.98	0.98	0.97	0.95	0.93	0.91	0.89	0.89
	global 1	0.82	0.82	0.79	0.76	0.68	0.58	0.47	0.40	0.33	0.27
	global 2	0.82	0.81	0.79	0.75	0.68	0.57	0.46	0.38	0.31	0.27
DLX 1.5	EDF	0.95	0.74	0.50	0.38	0.43	0.50	0.47	0.47	0.39	0.37
	partitioned	0.95	0.94	0.92	0.89	0.85	0.81	0.75	0.72	0.70	0.71
	global 1	0.93	0.92	0.90	0.87	0.82	0.76	0.67	0.62	0.59	0.59
	global 2	0.92	0.91	0.89	0.86	0.81	0.75	0.67	0.62	0.59	0.59
DLX 2	EDF	0.91	0.81	0.70	0.56	0.39	0.26	0.19	0.32	0.27	0.20
	partitioned	0.91	0.88	0.84	0.79	0.73	0.66	0.57	0.51	0.49	0.54
	global 1	0.91	0.88	0.84	0.78	0.72	0.65	0.55	0.47	0.44	0.48
	global 2	0.91	0.87	0.83	0.78	0.71	0.64	0.54	0.46	0.43	0.48
DLX 5	EDF	0.97	0.94	0.89	0.83	0.75	0.65	0.50	0.37	0.16	0.12
	partitioned	0.97	0.94	0.89	0.83	0.75	0.64	0.49	0.33	0.21	0.15
	global 1	0.97	0.94	0.89	0.83	0.75	0.64	0.48	0.31	0.18	0.13
	global 2	0.97	0.94	0.89	0.83	0.75	0.64	0.48	0.30	0.18	0.13

Table C.18: *Linux Experiment 1: average quickness, $U_{aperiodic}$ = 20%.*

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	EDF	0.96	0.53	0.68	0.75	0.74	0.69	0.65	0.70	0.76	0.82
	partitioned	0.96	0.96	0.96	0.95	0.95	0.93	0.91	0.89	0.87	0.88
	global 1	0.61	0.60	0.58	0.54	0.49	0.43	0.38	0.34	0.30	0.26
	global 2	0.60	0.59	0.56	0.52	0.47	0.41	0.36	0.32	0.28	0.26
DLX 1.5	EDF	0.87	0.69	0.48	0.37	0.44	0.49	0.46	0.46	0.38	0.40
	partitioned	0.87	0.86	0.84	0.82	0.79	0.75	0.70	0.68	0.67	0.68
	global 1	0.80	0.78	0.75	0.72	0.67	0.61	0.55	0.53	0.52	0.54
	global 2	0.79	0.76	0.73	0.70	0.66	0.61	0.56	0.53	0.53	0.54
DLX 2	EDF	0.79	0.70	0.62	0.51	0.37	0.26	0.20	0.32	0.22	0.17
	partitioned	0.79	0.75	0.71	0.67	0.63	0.58	0.51	0.46	0.46	0.51
	global 1	0.77	0.72	0.67	0.61	0.54	0.47	0.40	0.37	0.38	0.43
	global 2	0.75	0.71	0.65	0.59	0.52	0.46	0.40	0.36	0.38	0.44
DLX 5	EDF	0.89	0.83	0.74	0.65	0.55	0.45	0.36	0.29	0.15	0.09
	partitioned	0.89	0.82	0.74	0.64	0.52	0.40	0.30	0.22	0.17	0.14
	global 1	0.89	0.82	0.74	0.62	0.47	0.29	0.18	0.13	0.11	0.12
	global 2	0.89	0.82	0.73	0.61	0.45	0.28	0.17	0.13	0.11	0.12

Table C.19: *Linux Experiment 1: average quickness, $U_{aperiodic}$ = 50%.*



(a) *0% offline utilization.*

(b) *10% offline utilization.*

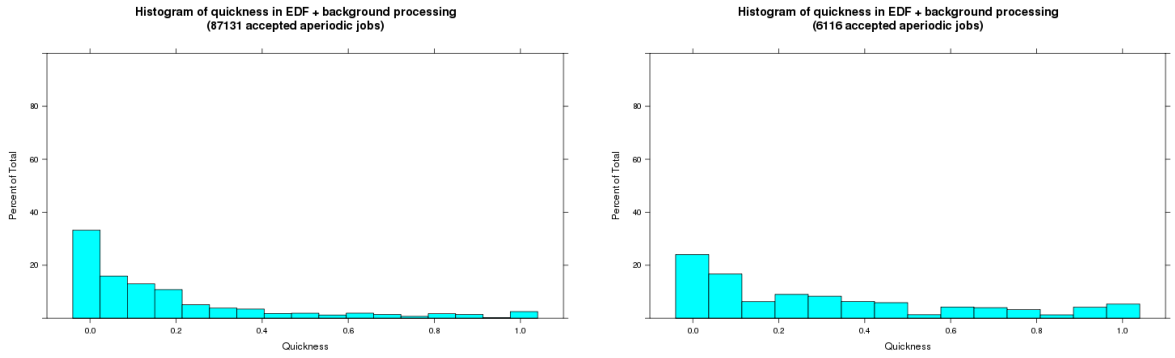
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

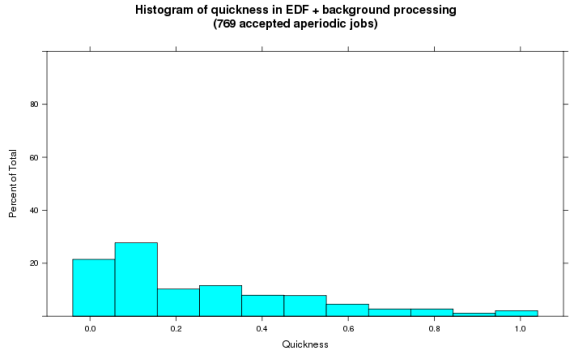
(f) *50% offline utilization.*

Figure C.1: *Linux Experiment 1: histograms of quickness of EDF with background processing of aperiodic jobs, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*

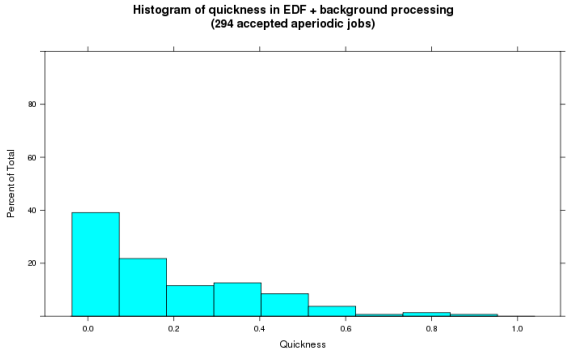


(a) *60% offline utilization.*

(b) *70% offline utilization.*

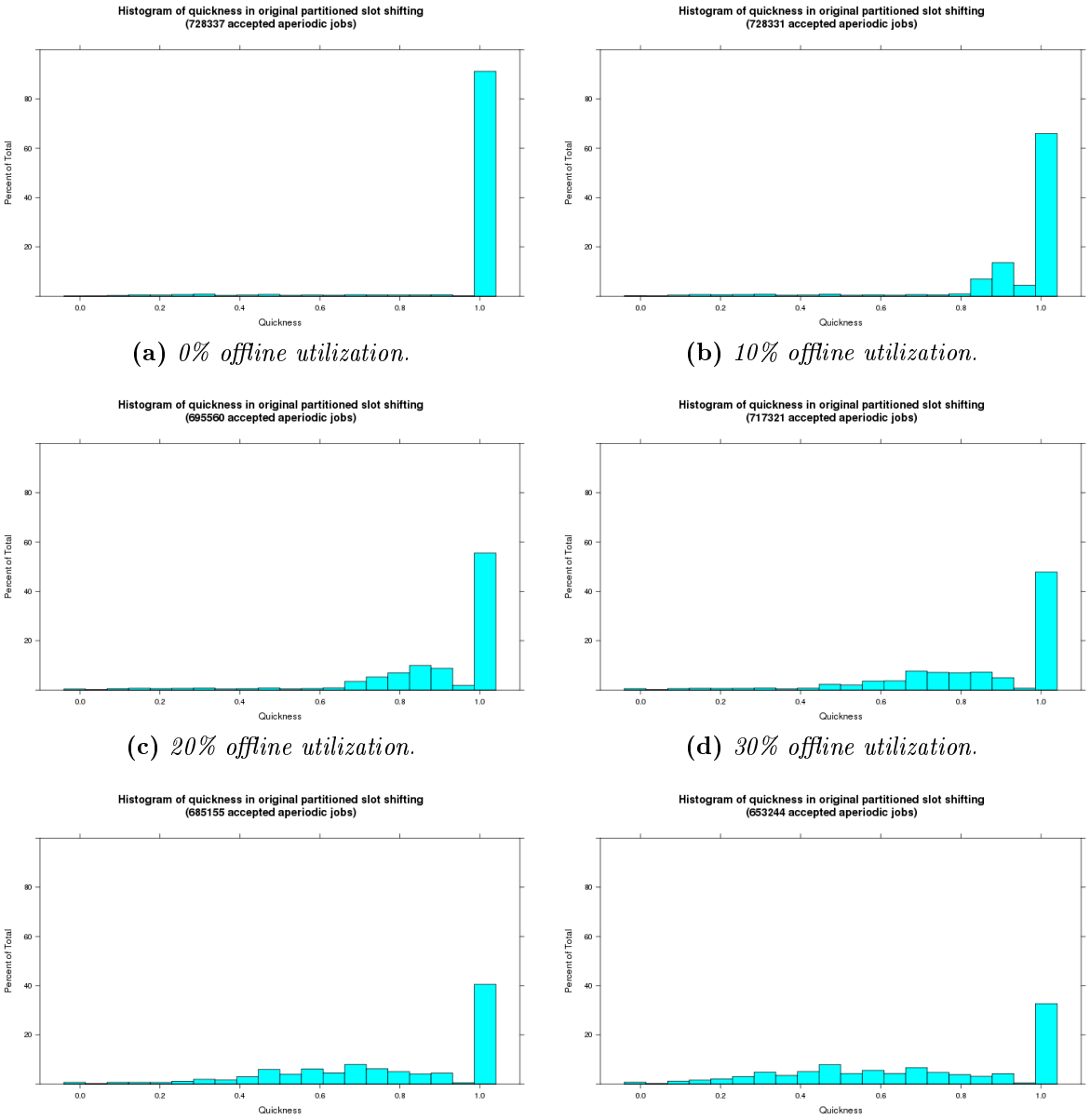


(c) *80% offline utilization.*



(d) *90% offline utilization.*

Figure C.2: *Linux Experiment 1: histograms of quickness of EDF with background processing of aperiodic jobs, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *0% offline utilization.*

(b) *10% offline utilization.*

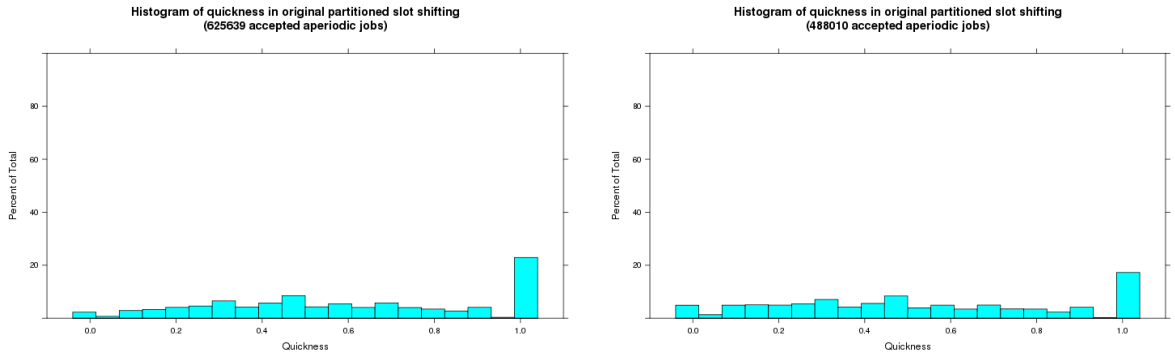
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

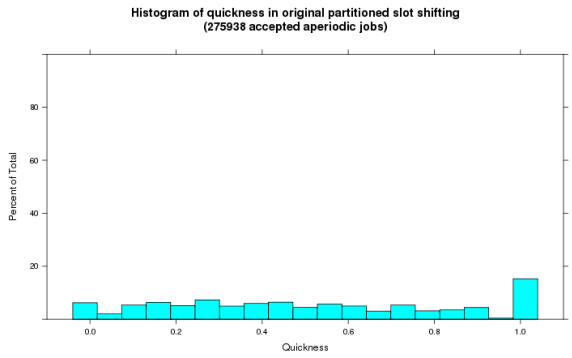
(f) *50% offline utilization.*

Figure C.3: *Linux Experiment 1: histograms of quickness of the partitioned slot shifting algorithm, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*

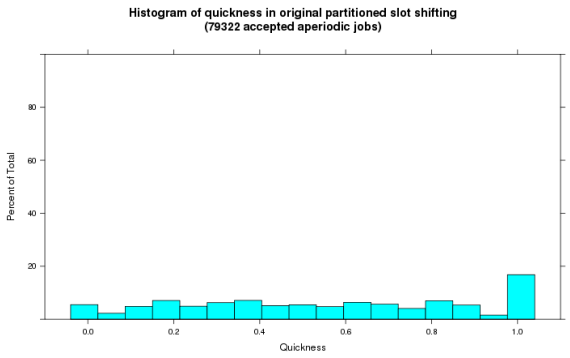


(a) *60% offline utilization.*

(b) *70% offline utilization.*

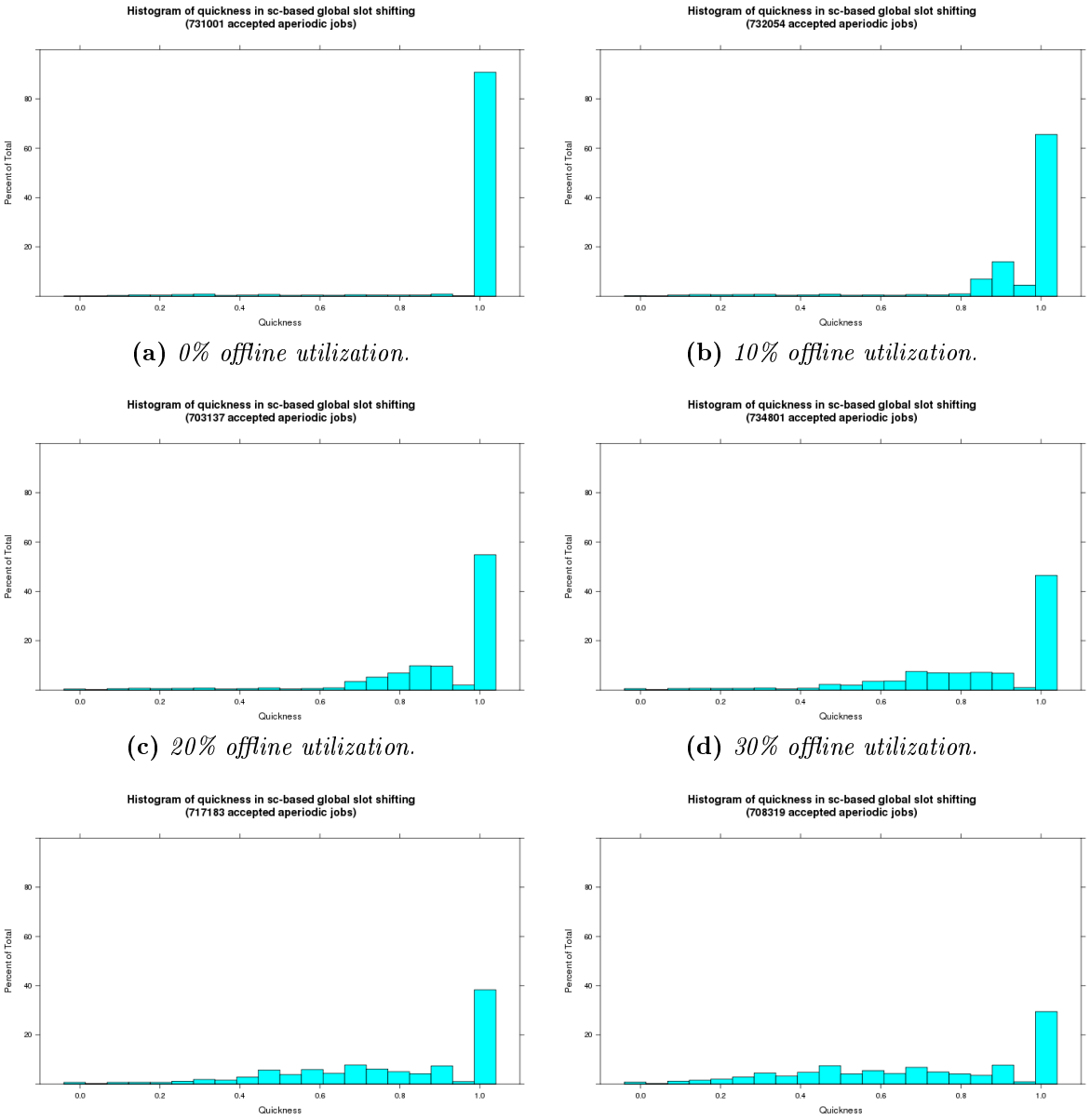


(c) *80% offline utilization.*



(d) *90% offline utilization.*

Figure C.4: *Linux Experiment 1: histograms of quickness of the partitioned slot shifting algorithm, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *0% offline utilization.*

(b) *10% offline utilization.*

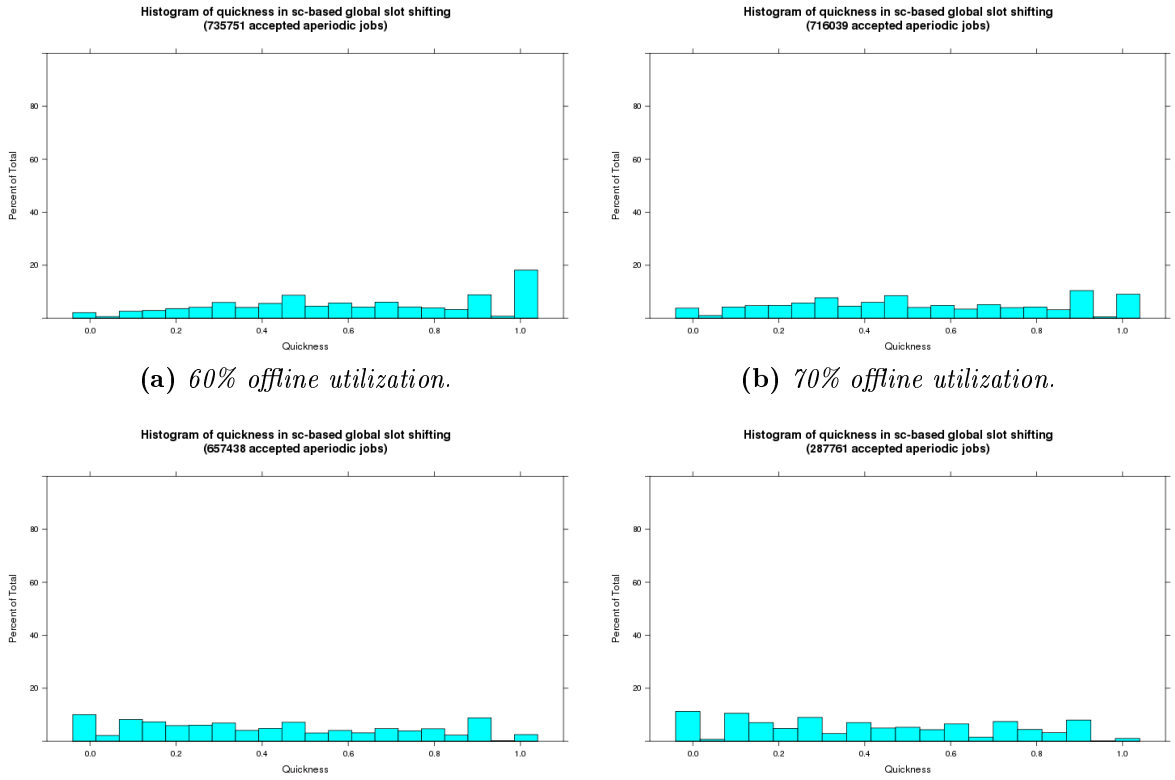
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

(f) *50% offline utilization.*

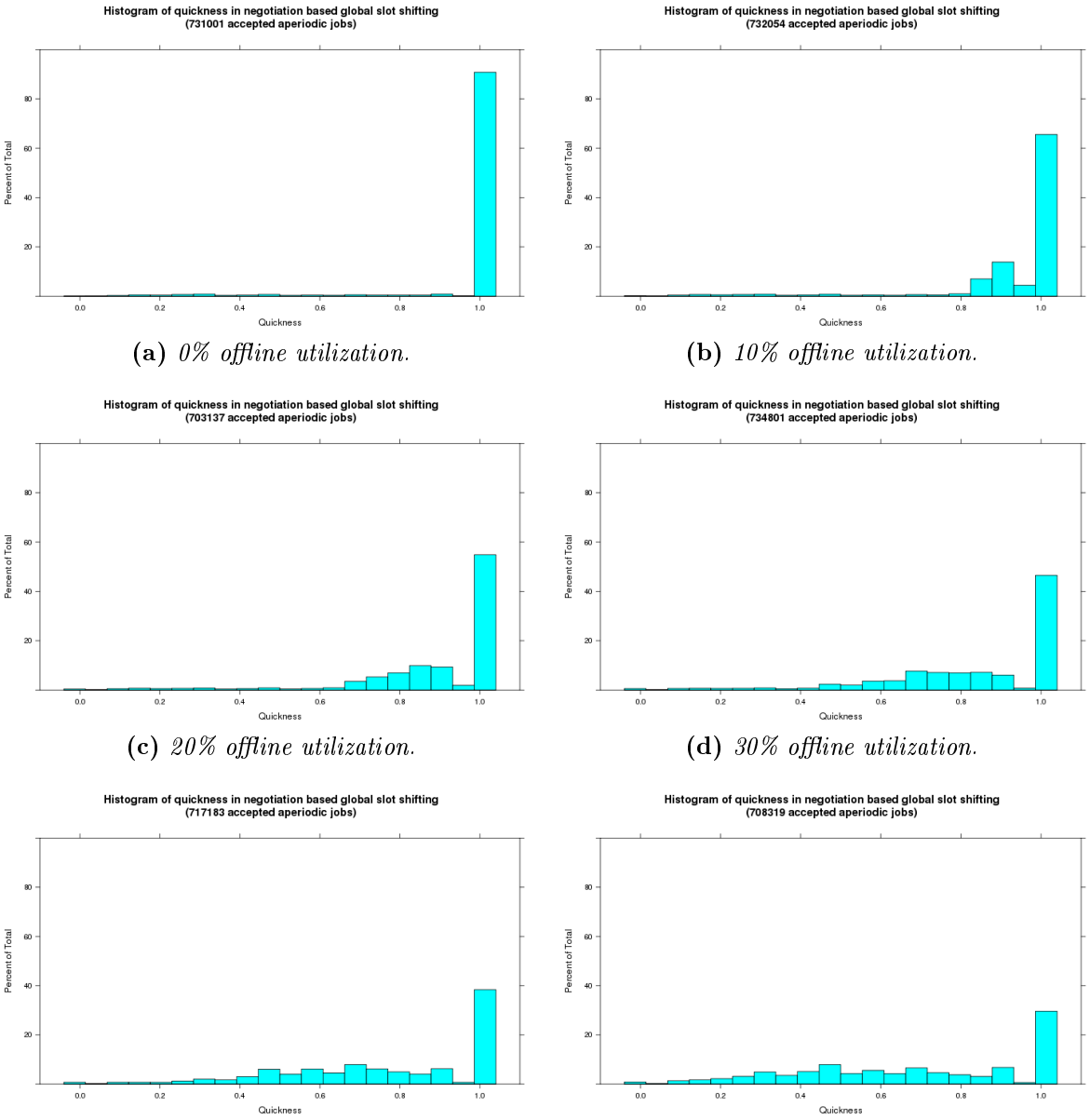
Figure C.5: *Linux Experiment 1: histograms of quickness of the spare-capacity-based slot shifting algorithm, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *60% offline utilization.* **(b)** *70% offline utilization.*

(c) *80% offline utilization.* **(d)** *90% offline utilization.*

Figure C.6: *Linux Experiment 1: histograms of quickness of the spare-capacity-based slot shifting algorithm, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *0% offline utilization.*

(b) *10% offline utilization.*

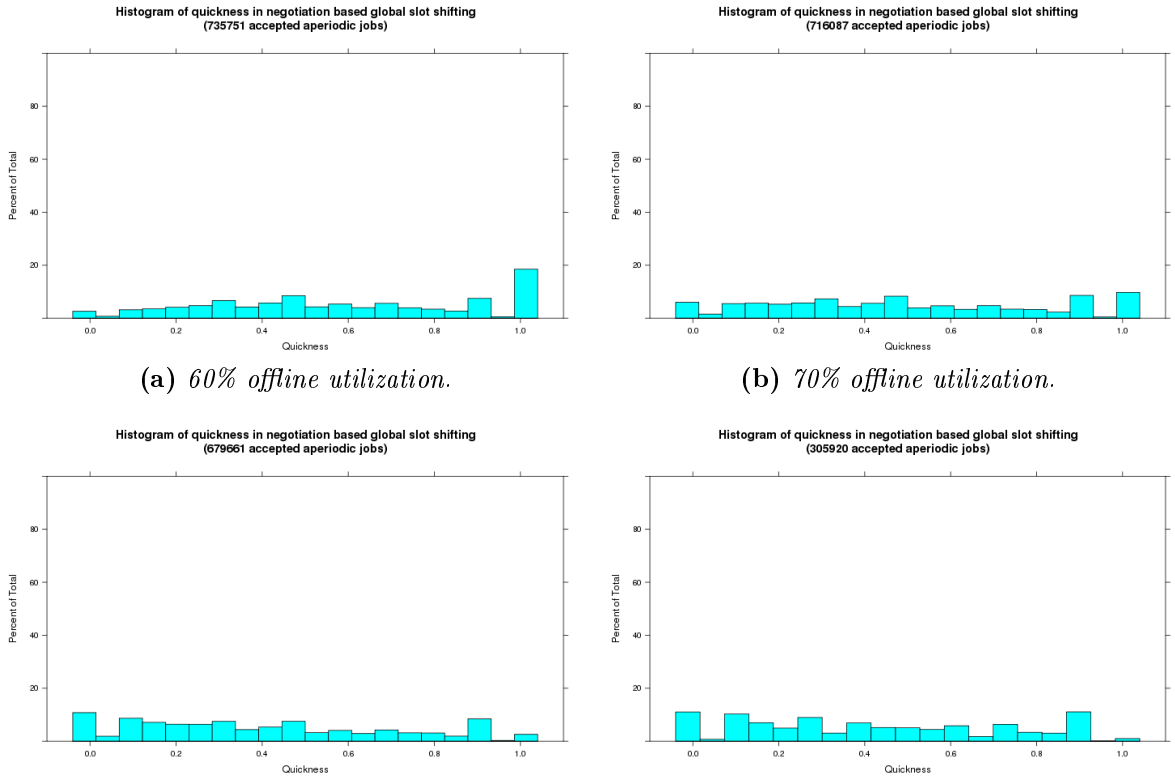
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

(f) *50% offline utilization.*

Figure C.7: *Linux Experiment 1: histograms of quickness of the negotiation-based slot shifting algorithm, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



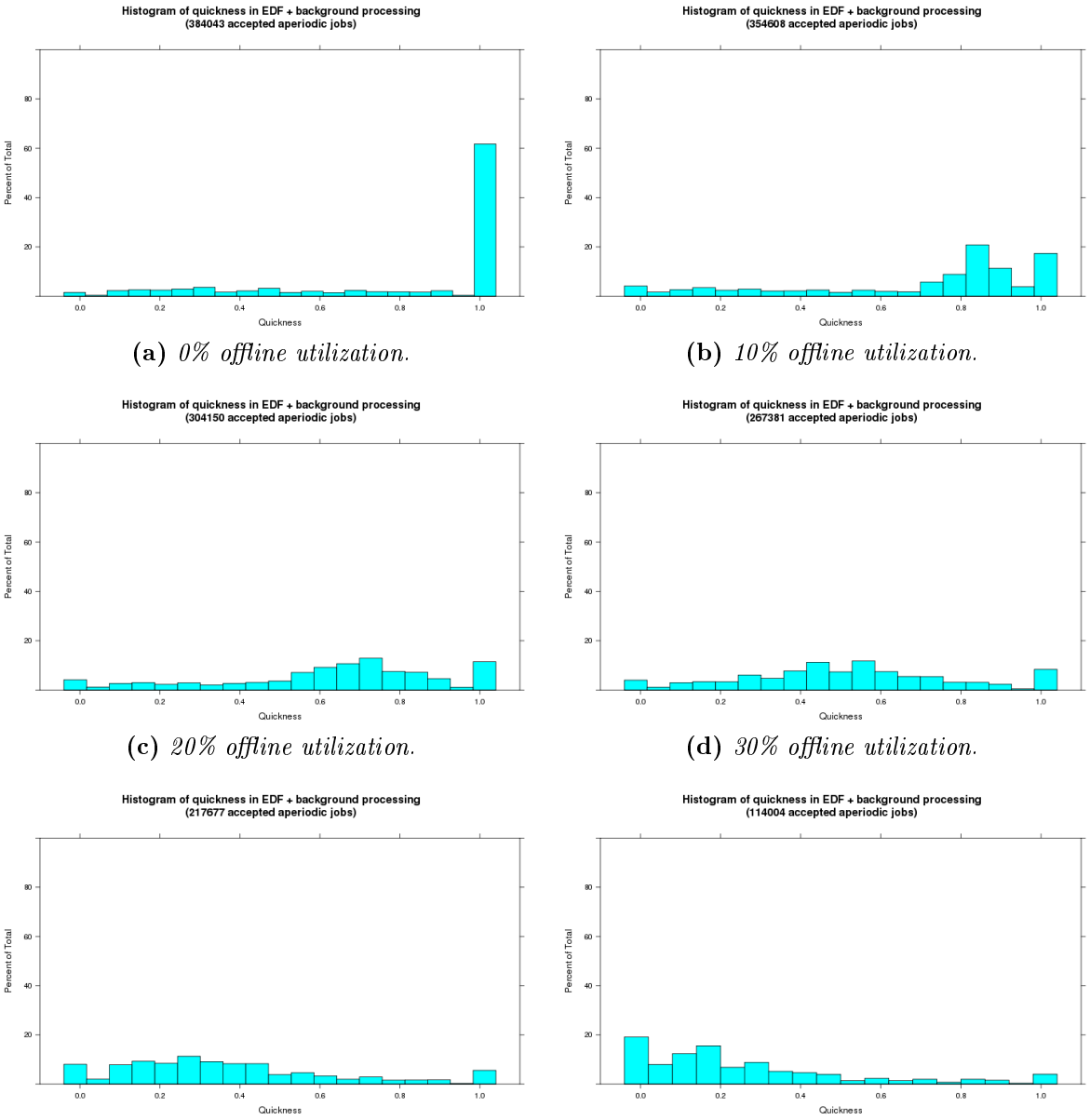
(a) *60% offline utilization.*

(b) *70% offline utilization.*

(c) *80% offline utilization.*

(d) *90% offline utilization.*

Figure C.8: *Linux Experiment 1: histograms of quickness of the negotiation-based slot shifting algorithm, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *0% offline utilization.*

(b) *10% offline utilization.*

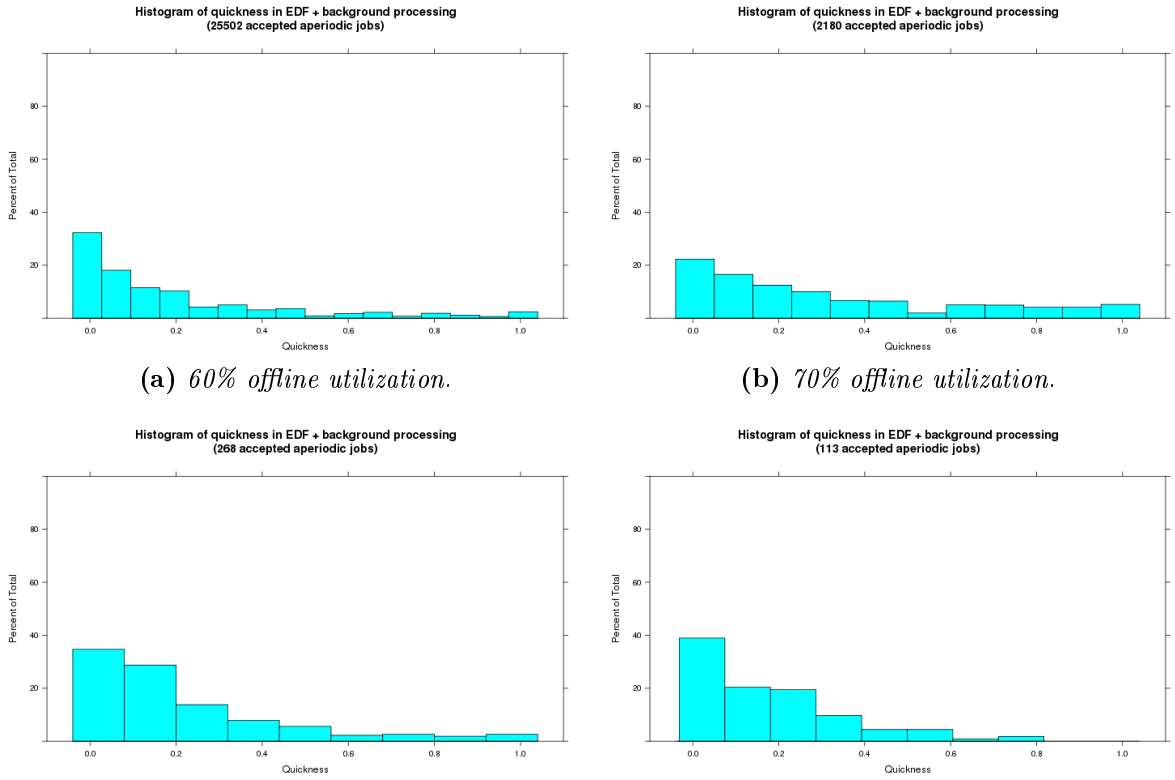
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

(f) *50% offline utilization.*

Figure C.9: *Linux Experiment 1: histograms of quickness of EDF with background processing of aperiodic jobs, 10% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



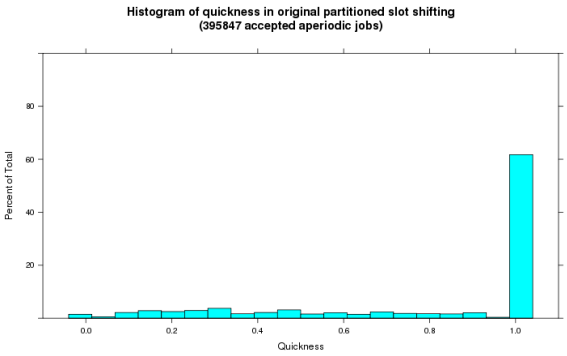
(a) *60% offline utilization.*

(b) *70% offline utilization.*

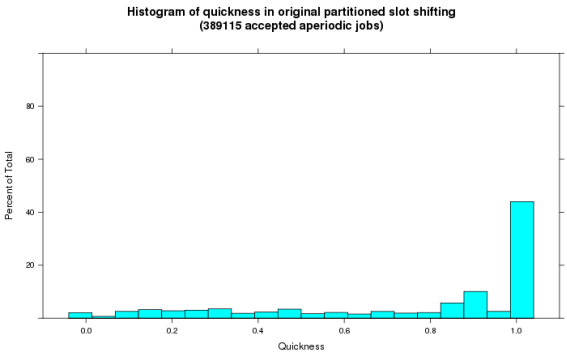
(c) *80% offline utilization.*

(d) *90% offline utilization.*

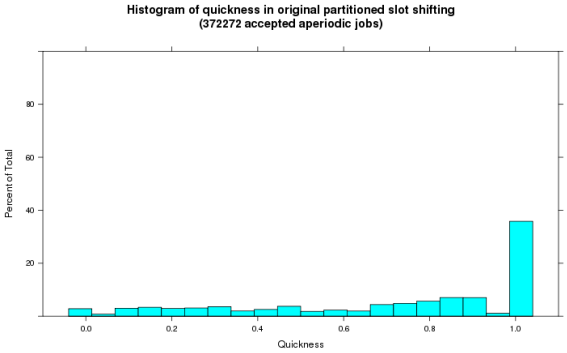
Figure C.10: *Linux Experiment 1: histograms of quickness of EDF with background processing of aperiodic jobs, 50% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



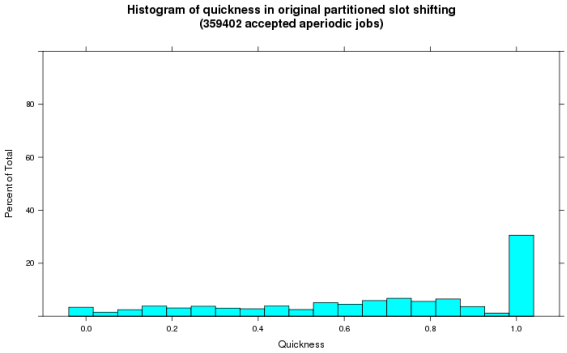
(a) *0% offline utilization.*



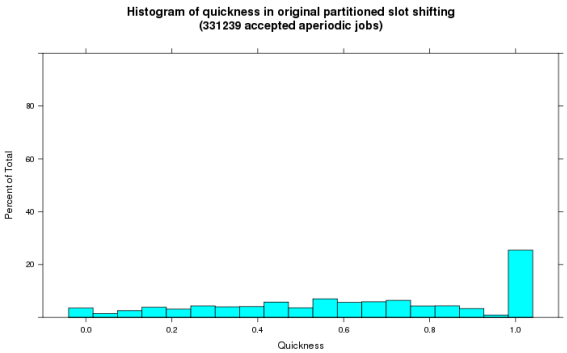
(b) *10% offline utilization.*



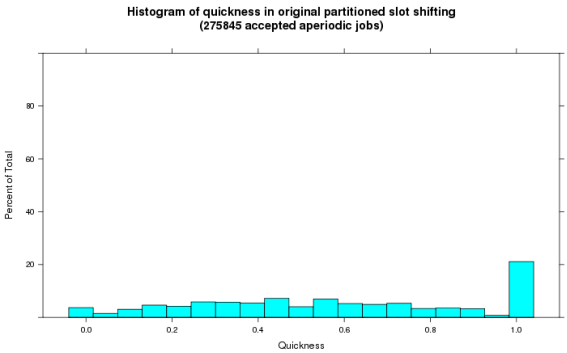
(c) *20% offline utilization.*



(d) *30% offline utilization.*

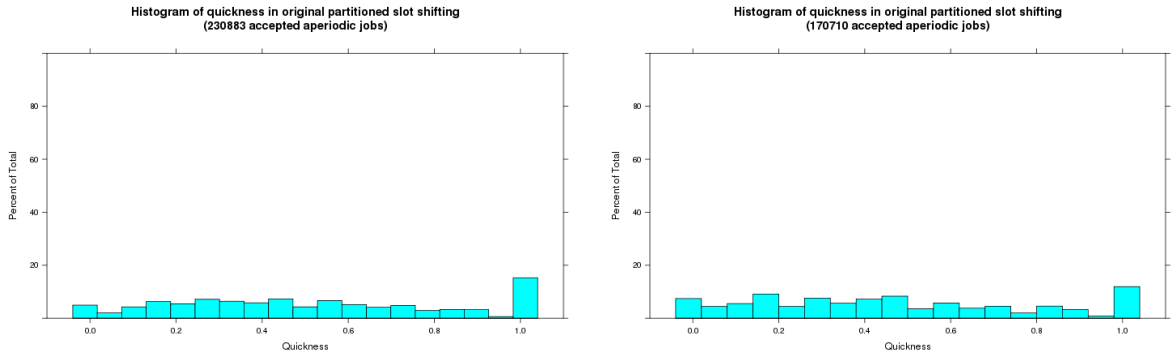


(e) *40% offline utilization.*



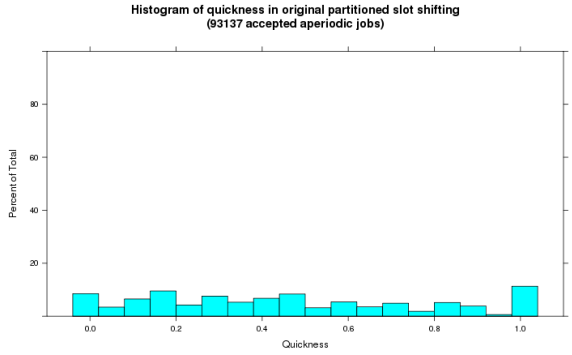
(f) *50% offline utilization.*

Figure C.11: *Linux Experiment 1: histograms of quickness of the partitioned slot shifting algorithm, 50% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*

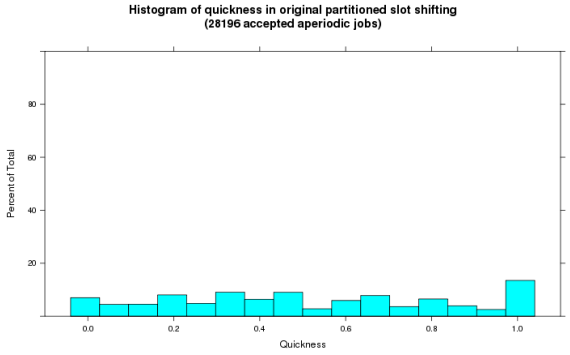


(a) *60% offline utilization.*

(b) *70% offline utilization.*

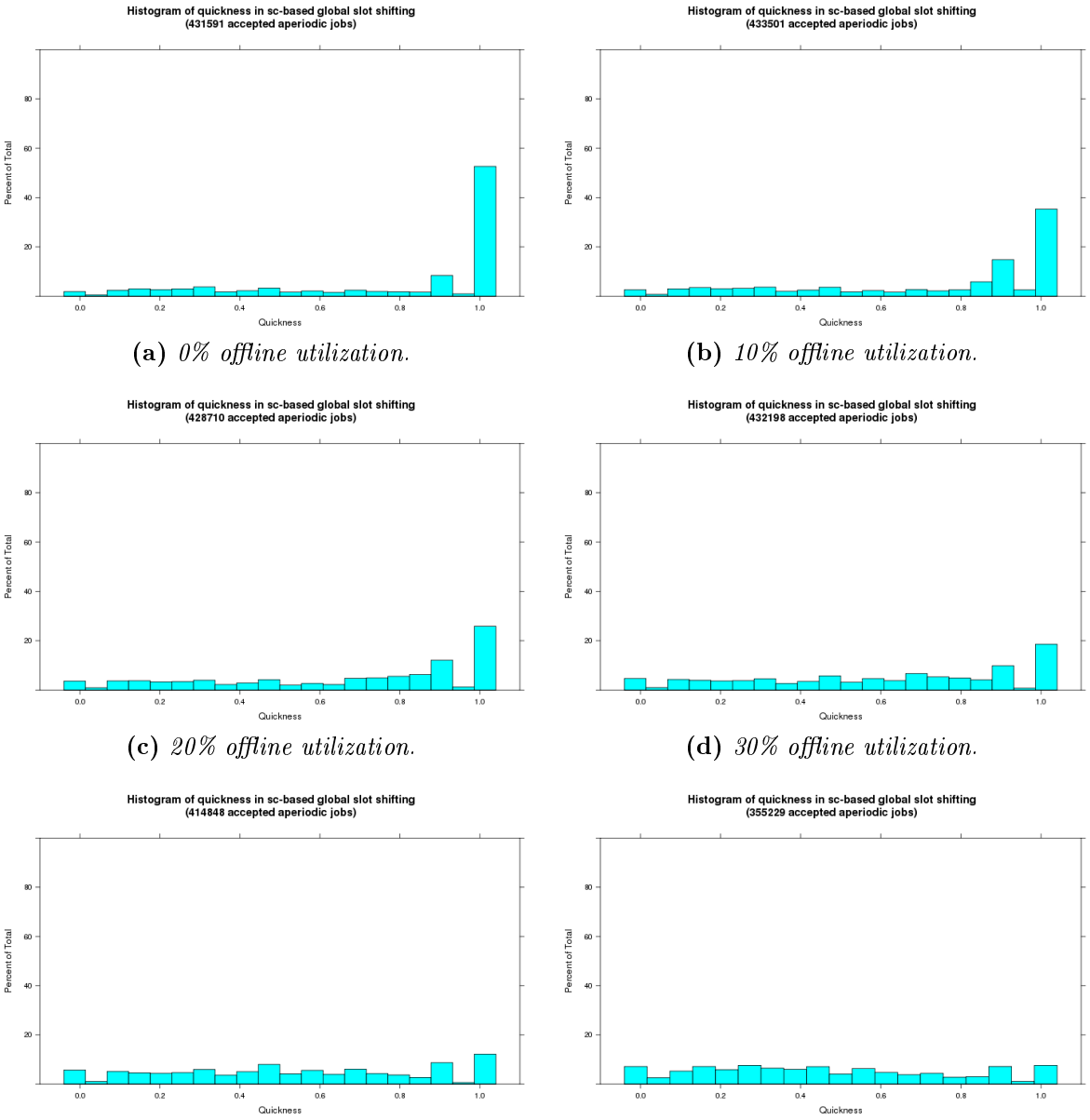


(c) *80% offline utilization.*



(d) *90% offline utilization.*

Figure C.12: *Linux Experiment 1: histograms of quickness of the partitioned slot shifting algorithm, 50% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *0% offline utilization.*

(b) *10% offline utilization.*

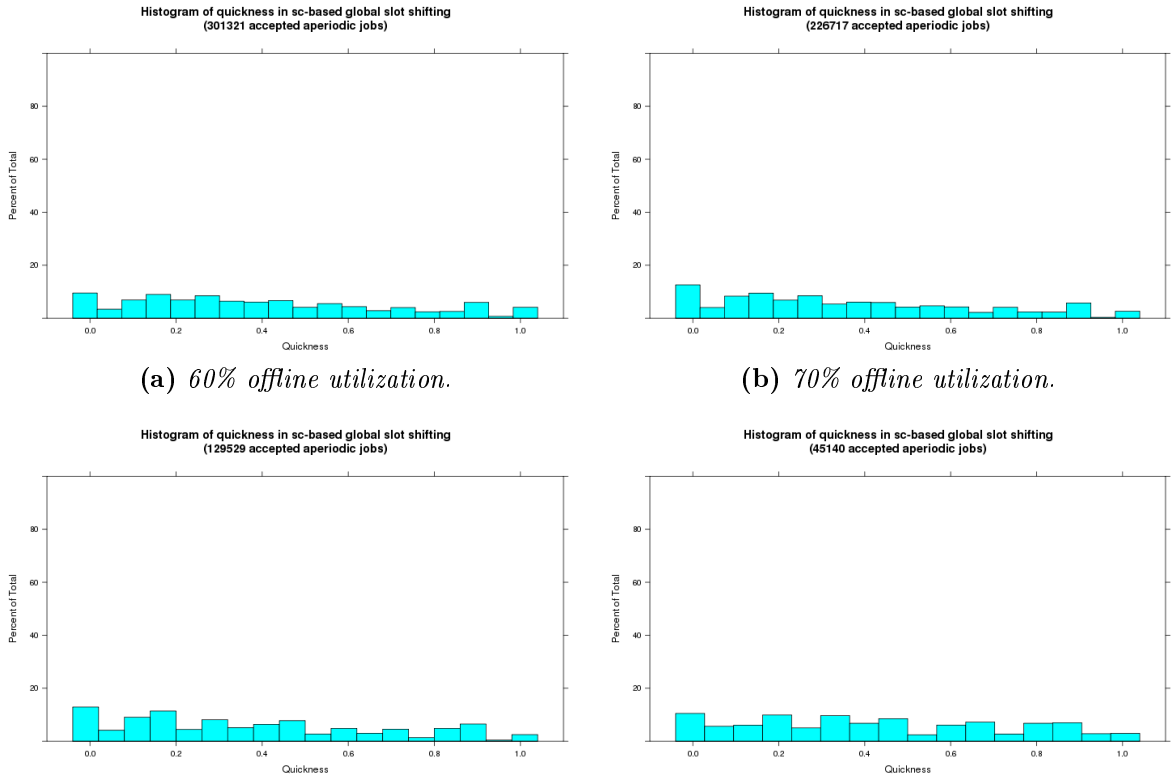
(c) *20% offline utilization.*

(d) *30% offline utilization.*

(e) *40% offline utilization.*

(f) *50% offline utilization.*

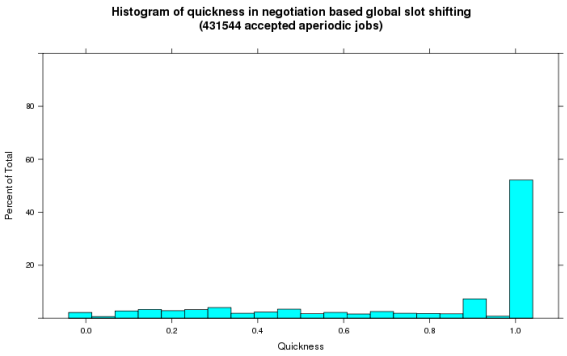
Figure C.13: *Linux Experiment 1: histograms of quickness of the spare-capacity-based slot shifting algorithm, 50% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



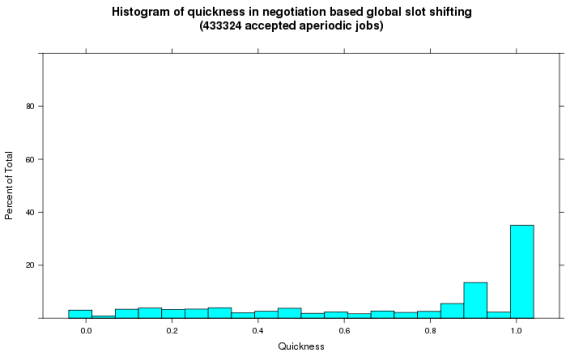
(a) *60% offline utilization.* **(b)** *70% offline utilization.*

(c) *80% offline utilization.* **(d)** *90% offline utilization.*

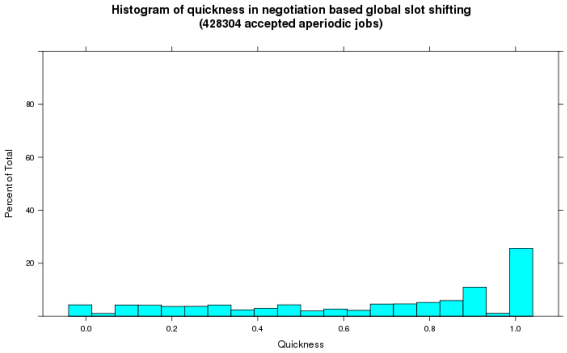
Figure C.14: *Linux Experiment 1: histograms of quickness of the spare-capacity-based slot shifting algorithm, 50% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



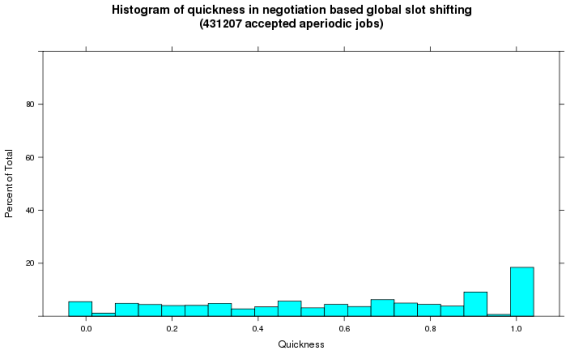
(a) *0% offline utilization.*



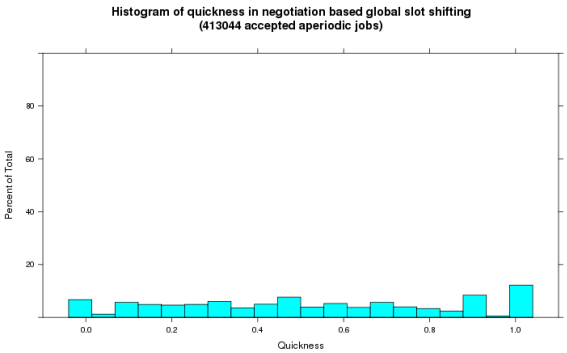
(b) *10% offline utilization.*



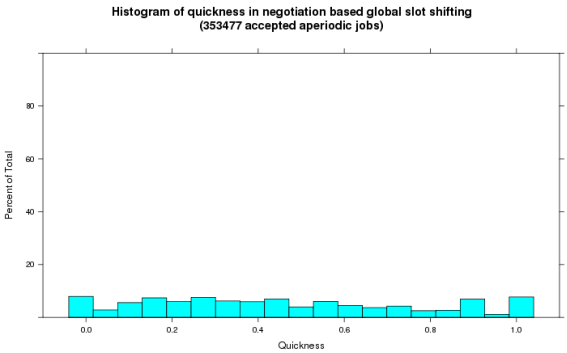
(c) *20% offline utilization.*



(d) *30% offline utilization.*

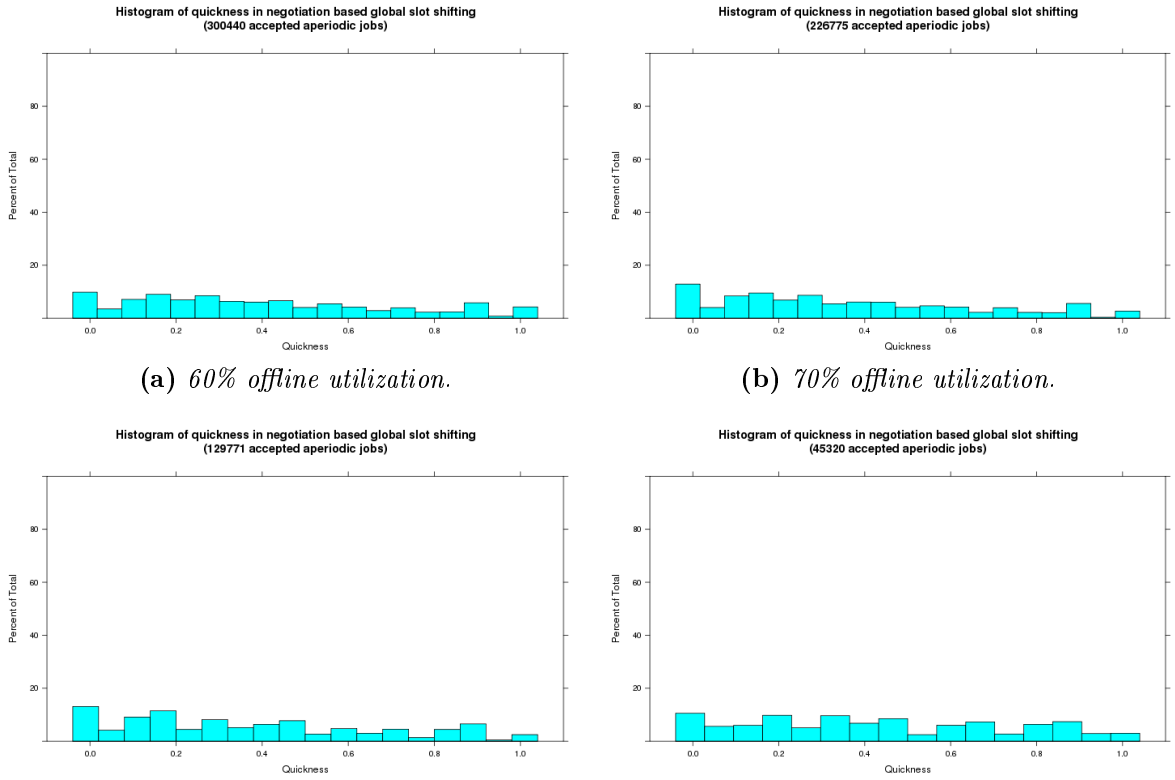


(e) *40% offline utilization.*



(f) *50% offline utilization.*

Figure C.15: *Linux Experiment 1: histograms of quickness of the negotiation-based slot shifting algorithm, 50% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*



(a) *60% offline utilization.*

(b) *70% offline utilization.*

(c) *80% offline utilization.*

(d) *90% offline utilization.*

Figure C.16: *Linux Experiment 1: histograms of quickness of the negotiation-based slot shifting algorithm, 50% aperiodic utilization, DLX factor 2. Note that jobs with a quickness value of 1 have a minimal response time; rejected jobs are filtered out.*

Appendix D

Detailed Results of Linux Experiment 2

This appendix lists the results of the effectiveness measurements we performed on Elwetritsch, the high performance cluster of the University of Kaiserslautern:
Experiment 2 - This experiment runs on a simulated 32-core system, the offline utilization on all cores is balanced, i.e., the same for all cores. The task parameters are similar to those in Experiment 1, listed in Table C.1.
All the following tables and figures list the results for three distinct scenarios which differ in the utilization created by the arriving aperiodic jobs: 10%, 20%, and 50%. The experiment has been conducted for offline utilizations between 0% and 90% in steps of 10%; each result being based on 1000 simulated jobs sets. Furthermore, each table lists the results for DLX factors of 1, 1.5, 2, and 5. The following scheduling algorithms have been applied:

- EDF with background processing of aperiodic jobs
- partitioned slot shifting
- global spare-capacity-based slot shifting (global algorithm 1)
- global negotiation-based slot shifting (global algorithm 2).

D.1 Acceptance Ratio

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 2	EDF	92.15	46.66	23.43	13.55	7.94	3.50	0.75	0.23	0.12	0.07
DLX 1	part.	92.49	92.09	91.01	88.54	82.24	69.61	52.44	35.60	19.09	5.94
	global 1	99.78	99.75	99.68	99.51	98.90	97.22	93.68	84.07	56.52	22.61
	global 2	100	100	100	100	100	100	99.86	95.80	70.25	28.23
Exp. 2	EDF	95.53	93.45	89.86	62.66	26.40	9.47	1.78	0.36	0.11	0.06
DLX 1.5	part.	95.80	95.41	94.67	93.45	91.02	82.69	68.27	48.81	26.83	7.99
	global 1	100	100	100	100	100	100	99.96	98.57	78.82	33.62
	global 2	100	100	100	100	100	100	99.99	99.61	84.61	36.03
Exp. 2	EDF	99.63	98.53	95.56	92.23	84.83	51.52	11.78	0.88	0.12	0.05
DLX 2	part.	99.65	99.51	98.95	97.65	95.58	92.28	85.03	67.88	38.37	10.82
	global 1	100	100	100	100	100	100	100	99.99	91.90	39.36
	global 2	100	100	100	100	100	100	100	100	95.16	41.85
Exp. 2	EDF	100	100	100	99.99	99.94	99.45	96.66	82.78	38.30	0.19
DLX 5	part.	100	100	100	100	99.99	99.90	99.21	95.55	80.36	35.52
	global 1	100	100	100	100	100	100	100	100	100	74.38
	global 2	100	100	100	100	100	100	100	100	100	76.22

Table D.1: *Linux Experiment 2: measured acceptance ratio (in %), 10% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 2	EDF	84.37	42.61	21.25	12.47	7.26	3.23	0.69	0.21	0.10	0.07
DLX 1	part.	85.69	85.14	83.63	80.67	74.23	62.82	46.66	31.42	16.71	5.09
	global 1	98.52	98.40	98.05	97.37	95.60	91.87	84.46	67.75	41.21	15.75
	global 2	100	100	100	100	100	99.99	98.53	83.69	53.20	19.41
Exp. 2	EDF	90.38	86.56	81.24	55.29	23.44	8.49	1.57	0.33	0.10	0.05
DLX 1.5	part.	91.50	90.76	89.44	87.19	83.46	74.58	60.25	42.53	23.16	6.84
	global 1	100	100	100	100	100	99.99	99.30	88.58	58.35	23.10
	global 2	100	100	100	100	100	100	99.97	94.60	64.11	24.67
Exp. 2	EDF	98.41	95.81	89.99	83.92	74.18	43.76	9.92	0.74	0.11	0.04
DLX 2	part.	98.58	98.05	96.74	94.16	90.24	84.43	74.80	57.47	31.74	9.00
	global 1	100	100	100	100	100	100	100	98.68	68.73	26.54
	global 2	100	100	100	100	100	100	100	99.61	72.15	27.94
Exp. 2	EDF	100	100	99.98	99.88	99.25	96.46	87.30	63.59	24.70	0.14
DLX 5	part.	100	100	99.99	99.96	99.78	98.90	95.43	85.24	63.58	25.78
	global 1	100	100	100	100	100	100	100	100	93.73	44.04
	global 2	100	100	100	100	100	100	100	100	92.98	44.49

Table D.2: *Linux Experiment 2: measured acceptance ratio (in %), 20% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 2	EDF	64.29	31.85	15.97	9.41	5.58	2.48	0.52	0.17	0.08	0.05
DLX 1	part.	70.31	69.27	67.09	63.47	57.34	47.87	35.48	23.55	12.43	3.79
	global 1	87.98	87.07	85.34	82.40	77.56	70.20	57.24	39.65	23.05	8.82
	global 2	100	99.99	99.96	99.66	97.23	88.17	71.16	50.28	29.78	10.62
Exp. 2	EDF	73.68	66.54	58.60	38.29	16.38	6.00	1.15	0.24	0.07	0.04
DLX 1.5	part.	79.69	78.00	75.28	71.18	65.50	56.53	44.33	30.58	16.43	4.93
	global 1	99.85	99.79	99.63	99.18	97.19	88.09	70.40	49.85	30.59	12.21
	global 2	100	100	100	99.98	99.38	92.76	76.65	55.92	33.43	12.86
Exp. 2	EDF	88.88	81.46	69.95	59.67	47.95	26.52	5.91	0.49	0.07	0.03
DLX 2	part.	91.54	89.35	85.63	80.26	73.15	64.42	53.41	39.02	21.10	6.24
	global 1	100	100	100	100	99.71	94.48	78.11	58.74	34.34	13.45
	global 2	100	100	100	100	99.86	94.69	78.54	59.49	35.23	13.83
Exp. 2	EDF	99.82	99.26	97.44	92.61	82.38	66.13	45.23	24.21	6.85	0.05
DLX 5	part.	99.86	99.55	98.67	96.32	91.07	81.91	68.74	53.00	35.49	14.47
	global 1	100	100	100	100	100	95.82	75.10	55.92	39.17	19.41
	global 2	100	100	100	100	100	95.30	75.18	55.92	39.09	19.49

Table D.3: *Linux Experiment 2: measured acceptance ratio (in %), 50% aperiodic job utilization.*

Offline Utilization	EDF with Backgr. Proc.	Partitioned Slot Shifting	Global Slot Shifting 1	Global Slot Shifting 2
0%	99.629 ± 0.017	99.651 ± 0.016	100.000 ± 0.000	100.000 ± 0.000
10%	98.535 ± 0.033	99.513 ± 0.018	100.000 ± 0.000	100.000 ± 0.000
20%	95.560 ± 0.060	98.953 ± 0.029	100.000 ± 0.000	100.000 ± 0.000
30%	92.227 ± 0.071	97.654 ± 0.039	100.000 ± 0.000	100.000 ± 0.000
40%	84.834 ± 0.097	95.585 ± 0.055	100.000 ± 0.000	100.000 ± 0.000
50%	51.518 ± 0.145	92.282 ± 0.072	100.000 ± 0.000	100.000 ± 0.000
60%	11.778 ± 0.095	85.029 ± 0.100	100.000 ± 0.000	100.000 ± 0.000
70%	0.880 ± 0.028	67.882 ± 0.216	99.986 ± 0.004	99.997 ± 0.002
80%	0.122 ± 0.011	38.375 ± 0.270	91.897 ± 0.246	95.157 ± 0.195
90%	0.046 ± 0.006	10.816 ± 0.139	39.352 ± 0.277	41.826 ± 0.295

Table D.4: *Experiment 2: acceptance ratios with 95% confidence interval (10% aperiodic job utilization, DLX factor 2).*

Offline Utilization	EDF with Backgr. Proc.	Partitioned Slot Shifting	Global Slot Shifting 1	Global Slot Shifting 2
0%	98.409 ± 0.028	98.582 ± 0.024	100.000 ± 0.000	100.000 ± 0.000
10%	95.814 ± 0.043	98.047 ± 0.027	100.000 ± 0.000	100.000 ± 0.000
20%	89.994 ± 0.063	96.737 ± 0.035	100.000 ± 0.000	100.000 ± 0.000
30%	83.916 ± 0.070	94.164 ± 0.045	100.000 ± 0.000	100.000 ± 0.000
40%	74.184 ± 0.086	90.244 ± 0.057	100.000 ± 0.000	100.000 ± 0.000
50%	43.756 ± 0.099	84.432 ± 0.064	100.000 ± 0.000	100.000 ± 0.000
60%	9.923 ± 0.061	74.803 ± 0.085	99.999 ± 0.001	100.000 ± 0.000
70%	0.740 ± 0.020	57.469 ± 0.170	98.682 ± 0.069	99.595 ± 0.031
80%	0.110 ± 0.008	31.740 ± 0.210	68.728 ± 0.290	72.157 ± 0.297
90%	0.044 ± 0.004	8.995 ± 0.111	26.549 ± 0.192	27.944 ± 0.198

Table D.5: *Experiment 2: acceptance ratios with 95% confidence interval (20% aperiodic job utilization, DLX factor 2).*

Offline Utilization	EDF with Backgr. Proc.	Partitioned Slot Shifting	Global Slot Shifting 1	Global Slot Shifting 2
0%	88.876 ± 0.045	91.542 ± 0.031	100.000 ± 0.000	100.000 ± 0.000
10%	81.463 ± 0.053	89.346 ± 0.034	100.000 ± 0.000	100.000 ± 0.000
20%	69.954 ± 0.052	85.630 ± 0.035	100.000 ± 0.000	100.000 ± 0.000
30%	59.666 ± 0.052	80.255 ± 0.040	99.995 ± 0.001	99.999 ± 0.000
40%	47.954 ± 0.054	73.148 ± 0.048	99.709 ± 0.014	99.857 ± 0.012
50%	26.521 ± 0.056	64.418 ± 0.048	94.463 ± 0.053	94.632 ± 0.054
60%	5.909 ± 0.034	53.407 ± 0.056	78.100 ± 0.056	78.543 ± 0.051
70%	0.491 ± 0.011	39.023 ± 0.100	58.741 ± 0.083	59.489 ± 0.058
80%	0.073 ± 0.004	21.100 ± 0.126	34.347 ± 0.157	35.232 ± 0.157
90%	0.032 ± 0.003	6.240 ± 0.065	13.448 ± 0.095	13.832 ± 0.098

Table D.6: *Experiment 2: acceptance ratios with 95% confidence interval (50% aperiodic job utilization, DLX factor 2).*

D.2 Number of Acceptance Tests

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.08	1.09	1.10	1.13	1.21	1.37	1.61	1.90	2.32	2.75
	glob. alg. 2	1.11	1.11	1.13	1.18	1.33	1.82	3.27	7.69	16.71	26.75
DLX 1.5	glob. alg. 1	1.05	1.05	1.06	1.07	1.10	1.20	1.40	1.84	3.54	6.31
	glob. alg. 2	1.05	1.06	1.07	1.09	1.13	1.32	1.89	4.10	12.65	25.10
DLX 2	glob. alg. 1	1.00	1.00	1.01	1.02	1.04	1.08	1.17	1.44	3.36	10.33
	glob. alg. 2	1.00	1.01	1.01	1.03	1.06	1.11	1.27	1.97	8.42	23.70
DLX 5	glob. alg. 1	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.25	10.12
	glob. alg. 2	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.06	1.49	15.90

Table D.7: *Average number of acceptance tests per aperiodic job for Experiment 2 (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	glob. alg. 2	–	–	–	–	–	32.00	32.00	32.00	32.00	32.00
DLX 1.5	glob. alg. 1	–	–	–	–	–	8.50	8.29	8.50	8.56	8.33
	glob. alg. 2	–	–	–	–	–	–	32.00	32.00	32.00	32.00
DLX 2	glob. alg. 1	–	–	–	–	–	–	–	15.85	16.17	15.40
	glob. alg. 2	–	–	–	–	–	–	–	32.00	32.00	32.00
DLX 5	glob. alg. 1	–	–	–	–	–	–	–	–	–	31.93
	glob. alg. 2	–	–	–	–	–	–	–	–	–	32.00

Table D.8: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 2 (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.08	1.08	1.10	1.12	1.19	1.33	1.51	1.69	1.80	1.89
	glob. alg. 2	1.11	1.11	1.13	1.18	1.33	1.82	3.23	6.64	10.29	13.46
DLX 1.5	glob. alg. 1	1.05	1.05	1.06	1.07	1.10	1.20	1.40	1.75	2.19	2.32
	glob. alg. 2	1.05	1.06	1.07	1.09	1.13	1.32	1.89	4.00	9.17	12.89
DLX 2	glob. alg. 1	1.00	1.00	1.01	1.02	1.04	1.08	1.17	1.44	2.24	2.54
	glob. alg. 2	1.00	1.01	1.01	1.03	1.06	1.11	1.27	1.97	7.25	12.20
DLX 5	glob. alg. 1	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.25	2.61
	glob. alg. 2	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.06	1.49	10.90

Table D.9: *Average number of acceptance tests per accepted aperiodic job for Experiment 2 (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.18	1.19	1.21	1.26	1.36	1.54	1.81	2.14	2.51	2.82
	glob. alg. 2	1.26	1.29	1.33	1.44	1.75	2.67	6.07	13.42	21.47	28.57
DLX 1.5	glob. alg. 1	1.10	1.11	1.13	1.16	1.22	1.36	1.72	2.88	4.93	6.88
	glob. alg. 2	1.13	1.15	1.17	1.23	1.35	1.76	3.14	9.03	19.02	27.57
DLX 2	glob. alg. 1	1.01	1.02	1.03	1.06	1.11	1.18	1.38	2.18	6.84	11.81
	glob. alg. 2	1.02	1.02	1.04	1.08	1.16	1.32	1.84	4.78	17.17	26.87
DLX 5	glob. alg. 1	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.18	4.24	19.25
	glob. alg. 2	1.00	1.00	1.00	1.00	1.00	1.01	1.06	1.39	9.91	23.85

Table D.10: *Average number of acceptance tests per aperiodic job for Experiment 2 (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	glob. alg. 2	–	–	–	–	–	32.00	32.00	32.00	32.00	32.00
DLX 1.5	glob. alg. 1	–	–	7.20	7.14	7.70	7.98	8.15	8.41	8.41	8.22
	glob. alg. 2	–	–	–	–	–	–	32.00	32.00	32.00	32.00
DLX 2	glob. alg. 1	–	–	–	–	–	–	15.89	15.98	15.70	15.10
	glob. alg. 2	–	–	–	–	–	–	–	32.00	32.00	32.00
DLX 5	glob. alg. 1	–	–	–	–	–	–	–	–	32.00	31.90
	glob. alg. 2	–	–	–	–	–	–	–	–	32.00	32.00

Table D.11: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 2 (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.16	1.16	1.18	1.21	1.29	1.41	1.59	1.73	1.81	1.89
	glob. alg. 2	1.26	1.29	1.33	1.44	1.75	2.67	5.70	9.83	12.22	14.39
DLX 1.5	glob. alg. 1	1.10	1.11	1.13	1.16	1.22	1.36	1.68	2.17	2.45	2.41
	glob. alg. 2	1.13	1.15	1.17	1.23	1.35	1.76	3.13	7.75	11.78	14.07
DLX 2	glob. alg. 1	1.01	1.02	1.03	1.06	1.11	1.18	1.38	2.00	2.82	2.74
	glob. alg. 2	1.02	1.02	1.04	1.08	1.16	1.32	1.84	4.68	11.48	13.67
DLX 5	glob. alg. 1	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.18	2.38	3.17
	glob. alg. 2	1.00	1.00	1.00	1.00	1.00	1.01	1.06	1.39	8.25	13.68

Table D.12: *Average number of acceptance tests per accepted aperiodic job for Experiment 2 (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.50	1.53	1.58	1.67	1.80	1.99	2.24	2.50	2.72	2.90
	glob. alg. 2	2.71	2.96	3.54	4.87	8.06	12.79	17.82	22.60	26.78	30.26
DLX 1.5	glob. alg. 1	1.39	1.44	1.54	1.70	2.07	2.92	4.16	5.45	6.52	7.42
	glob. alg. 2	1.67	1.82	2.17	2.94	5.34	11.02	16.76	21.61	26.13	29.86
DLX 2	glob. alg. 1	1.10	1.13	1.19	1.31	1.70	3.21	5.76	8.22	11.04	13.19
	glob. alg. 2	1.15	1.23	1.39	1.87	3.49	9.61	16.46	21.07	25.87	29.67
DLX 5	glob. alg. 1	1.00	1.00	1.01	1.04	1.10	3.09	10.11	15.84	20.72	26.39
	glob. alg. 2	1.00	1.01	1.02	1.06	1.28	6.57	15.99	21.52	25.44	28.81

Table D.13: *Average number of acceptance tests per aperiodic job for Experiment 2 (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	glob. alg. 2	32.00	32.00	32.00	32.00	32.00	32.00	32.00	32.00	32.00	32.00
DLX 1.5	glob. alg. 1	7.48	7.49	7.57	7.68	7.84	8.00	8.13	8.22	8.20	8.11
	glob. alg. 2	–	–	32.00	32.00	32.00	32.00	32.00	32.00	32.00	32.00
DLX 2	glob. alg. 1	–	–	–	13.67	13.94	14.40	14.94	15.15	15.05	14.79
	glob. alg. 2	–	–	–	32.00	32.00	32.00	32.00	32.00	32.00	32.00
DLX 5	glob. alg. 1	–	–	–	–	–	31.99	31.99	32.00	32.00	31.87
	glob. alg. 2	–	–	–	–	–	32.00	32.00	32.00	32.00	32.00

Table D.14: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 2 (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	glob. alg. 1	1.30	1.31	1.34	1.38	1.45	1.56	1.67	1.73	1.79	1.88
	glob. alg. 2	2.71	2.95	3.53	4.78	7.38	10.21	12.07	13.31	14.48	15.66
DLX 1.5	glob. alg. 1	1.38	1.43	1.51	1.66	1.90	2.24	2.49	2.66	2.72	2.49
	glob. alg. 2	1.67	1.82	2.17	2.94	5.18	9.39	12.12	13.40	14.46	15.34
DLX 2	glob. alg. 1	1.10	1.13	1.19	1.31	1.67	2.55	3.18	3.36	3.39	2.94
	glob. alg. 2	1.15	1.23	1.39	1.87	3.45	8.34	12.21	13.63	14.60	15.21
DLX 5	glob. alg. 1	1.00	1.00	1.01	1.04	1.10	1.84	2.86	3.11	3.21	3.64
	glob. alg. 2	1.00	1.01	1.02	1.06	1.28	5.33	10.70	13.25	15.22	15.62

Table D.15: *Average number of acceptance tests per accepted aperiodic job for Experiment 2 (50% aperiodic job utilization).*

D.3 Quickness

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	EDF	0.99	0.54	0.70	0.77	0.75	0.70	0.67	0.71	0.79	0.78
	partitioned	0.99	0.99	0.99	0.98	0.98	0.96	0.94	0.92	0.90	0.91
	global 1	0.91	0.91	0.89	0.86	0.79	0.64	0.45	0.28	0.17	0.10
	global 2	0.91	0.90	0.89	0.86	0.78	0.62	0.42	0.23	0.12	0.06
DLX 1.5	EDF	0.97	0.76	0.51	0.38	0.44	0.50	0.47	0.48	0.43	0.39
	partitioned	0.97	0.96	0.95	0.92	0.88	0.83	0.77	0.73	0.72	0.73
	global 1	0.97	0.95	0.94	0.91	0.87	0.82	0.75	0.63	0.48	0.46
	global 2	0.96	0.95	0.94	0.91	0.86	0.80	0.71	0.62	0.53	0.51
DLX 2	EDF	0.96	0.85	0.73	0.58	0.40	0.27	0.19	0.31	0.25	0.19
	partitioned	0.96	0.93	0.89	0.83	0.77	0.69	0.60	0.53	0.51	0.55
	global 1	0.96	0.93	0.89	0.84	0.77	0.70	0.61	0.53	0.42	0.41
	global 2	0.95	0.92	0.88	0.83	0.77	0.69	0.59	0.50	0.40	0.42
DLX 5	EDF	0.99	0.96	0.92	0.87	0.81	0.71	0.56	0.40	0.16	0.13
	partitioned	0.99	0.96	0.92	0.87	0.81	0.71	0.56	0.39	0.23	0.16
	global 1	0.99	0.96	0.92	0.87	0.81	0.71	0.57	0.39	0.24	0.12
	global 2	0.99	0.96	0.92	0.87	0.81	0.71	0.56	0.39	0.22	0.12

Table D.16: *Linux Experiment 2: average quickness,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	EDF	0.98	0.53	0.69	0.76	0.75	0.70	0.66	0.74	0.81	0.79
	partitioned	0.98	0.98	0.98	0.98	0.97	0.95	0.93	0.91	0.89	0.90
	global 1	0.83	0.82	0.80	0.76	0.69	0.56	0.37	0.23	0.16	0.09
	global 2	0.81	0.81	0.78	0.74	0.65	0.49	0.28	0.14	0.08	0.05
DLX 1.5	EDF	0.95	0.74	0.50	0.38	0.44	0.50	0.48	0.48	0.45	0.38
	partitioned	0.95	0.93	0.92	0.89	0.85	0.81	0.75	0.72	0.70	0.72
	global 1	0.93	0.92	0.90	0.87	0.83	0.78	0.65	0.47	0.39	0.40
	global 2	0.92	0.91	0.89	0.86	0.81	0.74	0.64	0.52	0.45	0.45
DLX 2	EDF	0.91	0.81	0.70	0.57	0.39	0.26	0.19	0.31	0.24	0.17
	partitioned	0.91	0.88	0.84	0.79	0.73	0.66	0.57	0.51	0.49	0.54
	global 1	0.91	0.88	0.84	0.79	0.73	0.66	0.56	0.44	0.35	0.35
	global 2	0.91	0.87	0.83	0.78	0.72	0.63	0.53	0.41	0.32	0.37
DLX 5	EDF	0.97	0.94	0.89	0.83	0.75	0.65	0.50	0.37	0.16	0.12
	partitioned	0.97	0.93	0.89	0.83	0.75	0.64	0.49	0.33	0.21	0.15
	global 1	0.97	0.93	0.89	0.83	0.75	0.64	0.49	0.33	0.14	0.10
	global 2	0.97	0.93	0.89	0.83	0.75	0.64	0.48	0.30	0.12	0.10

Table D.17: *Linux Experiment 2: average quickness,* $U_{aperiodic} = 20\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	EDF	0.96	0.53	0.68	0.75	0.74	0.69	0.65	0.71	0.77	0.74
	partitioned	0.96	0.96	0.96	0.95	0.95	0.93	0.91	0.89	0.87	0.88
	global 1	0.66	0.65	0.62	0.57	0.50	0.39	0.27	0.21	0.16	0.08
	global 2	0.52	0.49	0.44	0.35	0.23	0.14	0.09	0.07	0.05	0.04
DLX 1.5	EDF	0.87	0.69	0.48	0.37	0.43	0.48	0.47	0.46	0.40	0.37
	partitioned	0.87	0.86	0.84	0.82	0.79	0.75	0.70	0.68	0.67	0.68
	global 1	0.80	0.78	0.76	0.72	0.63	0.46	0.34	0.29	0.26	0.29
	global 2	0.78	0.75	0.71	0.65	0.54	0.40	0.33	0.31	0.32	0.33
DLX 2	EDF	0.79	0.70	0.62	0.51	0.37	0.26	0.20	0.31	0.24	0.17
	partitioned	0.79	0.75	0.71	0.67	0.63	0.57	0.51	0.46	0.46	0.51
	global 1	0.77	0.73	0.69	0.63	0.54	0.38	0.27	0.25	0.24	0.25
	global 2	0.75	0.70	0.64	0.57	0.45	0.29	0.21	0.19	0.21	0.26
DLX 5	EDF	0.89	0.82	0.74	0.65	0.55	0.45	0.36	0.29	0.15	0.11
	partitioned	0.89	0.82	0.74	0.64	0.52	0.40	0.30	0.22	0.17	0.14
	global 1	0.89	0.82	0.74	0.64	0.51	0.21	0.09	0.06	0.05	0.07
	global 2	0.89	0.82	0.73	0.61	0.44	0.16	0.07	0.05	0.05	0.07

Table D.18: *Linux Experiment 2: average quickness,* $U_{aperiodic} = 50\%$.

Appendix E

Detailed Results of Linux Experiment 3

This appendix lists the results of the effectiveness measurements we performed on Elwetritsch, the high performance cluster of the University of Kaiserslautern: **Experiment 3** - This experiment repeats Experiment 1and Experiment 2using SDL to improve the response time of the aperiodic jobs.
All the following tables and figures list the results for three distinct scenarios which differ in the utilization created by the arriving aperiodic jobs: 10%, 20%, and 50%. The experiment has been conducted for offline utilizations between 0% and 90% in steps of 10%; each result being based on 1000 simulated jobs sets. Furthermore, each table lists the results for DLX factors of 1, 1.5, 2, and 5. The following scheduling algorithms have been applied:

- EDF with background processing of aperiodic jobs
- partitioned slot shifting
- global spare-capacity-based slot shifting (global algorithm 1)
- global negotiation-based slot shifting (global algorithm 2).

E.1 Acceptance Ratio

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3a	EDF	92.27	46.85	23.24	13.43	7.85	3.58	0.71	0.23	0.10	0.07
DLX 1	part.	92.61	92.16	90.97	88.46	81.94	69.61	51.86	35.43	19.15	5.65
	global 1	99.91	99.88	99.84	99.70	98.76	94.82	82.76	63.73	38.38	12.66
	global 2	99.97	99.96	99.92	99.81	99.06	95.63	84.28	65.65	39.76	12.98
Exp. 3a	EDF	95.57	93.65	90.12	62.75	26.38	9.43	1.68	0.37	0.09	0.05
DLX 1.5	part.	95.81	95.52	94.74	93.58	91.21	82.77	67.42	48.89	26.82	8.19
	global 1	99.99	100	99.99	99.98	99.93	99.37	94.83	80.21	52.24	18.29
	global 2	99.99	100	99.99	99.98	99.93	99.40	95.08	80.79	52.68	18.43
Exp. 3a	EDF	99.67	98.55	95.68	92.33	85.17	51.92	11.81	0.85	0.11	0.04
DLX 2	part.	99.68	99.16	98.59	97.48	95.71	92.09	84.93	67.75	38.12	10.73
	global 1	100	100	100	100	99.99	99.95	99.50	94.24	66.65	22.83
	global 2	100	100	100	100	99.99	99.95	99.53	94.45	67.00	23.04
Exp. 3a	EDF	100	100	100	99.99	99.93	99.52	96.70	82.55	38.47	0.19
DLX 5	part.	100	100	100	99.99	99.96	99.85	99.06	94.89	80.22	35.17
	global 1	100	100	100	100	100	100	100	99.98	98.07	55.81
	global 2	100	100	100	100	100	100	100	99.98	97.96	55.97

Table E.1: *Measured acceptance ratio (in %) for Experiment 3a (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3a	EDF	84.57	42.70	21.11	12.45	7.30	3.24	0.72	0.19	0.11	0.07
DLX 1	part.	85.86	85.16	83.60	80.54	74.16	62.82	47.21	32.38	16.73	5.10
	global 1	99.29	99.14	98.87	98.22	96.04	89.39	74.70	54.84	30.80	10.04
	global 2	99.54	99.43	99.22	98.64	96.76	90.63	76.36	56.53	31.81	10.24
Exp. 3a	EDF	90.53	86.60	81.28	55.28	23.16	8.50	1.58	0.34	0.12	0.05
DLX 1.5	part.	91.61	90.76	89.43	87.14	83.27	74.55	60.35	42.46	23.52	6.80
	global 1	99.94	99.90	99.86	99.66	99.15	96.76	88.20	68.81	41.72	13.57
	global 2	99.94	99.90	99.85	99.67	99.12	96.79	88.42	69.34	42.02	13.63
Exp. 3a	EDF	98.48	95.71	89.94	84.06	74.22	43.83	9.84	0.72	0.09	0.05
DLX 2	part.	98.63	97.36	96.02	93.88	90.07	84.36	74.74	57.72	31.62	9.06
	global 1	100	100	99.99	99.97	99.86	99.25	96.07	83.26	51.34	17.09
	global 2	100	100	99.99	99.97	99.85	99.19	96.03	83.43	51.61	17.20
Exp. 3a	EDF	100	100	99.99	99.87	99.31	96.42	87.30	63.62	24.69	0.13
DLX 5	part.	100	100	99.98	99.92	99.69	98.63	94.98	84.72	63.37	25.70
	global 1	100	100	100	100	100	100	99.96	98.36	81.50	36.40
	global 2	100	100	100	100	100	100	99.95	98.19	81.11	36.50

Table E.2: *Measured acceptance ratio (in %) for Experiment 3a (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3a	EDF	64.34	32.00	15.90	9.38	5.52	2.47	0.54	0.17	0.08	0.05
DLX 1	part.	70.30	69.25	67.18	63.49	57.46	47.77	35.65	23.48	12.56	3.80
	global 1	92.00	91.03	88.89	84.96	78.53	67.79	52.60	35.48	19.85	6.35
	global 2	92.75	91.80	89.72	85.90	79.63	69.11	53.83	36.44	20.36	6.45
Exp. 3a	EDF	73.64	66.43	58.55	38.24	16.51	6.22	1.14	0.25	0.07	0.04
DLX 1.5	part.	79.63	77.98	75.23	71.14	65.51	56.61	44.09	30.57	16.36	4.94
	global 1	97.70	96.94	95.35	92.27	86.75	77.28	62.51	44.41	25.17	8.40
	global 2	97.62	96.80	95.12	91.92	86.41	77.13	62.56	44.63	25.28	8.43
Exp. 3a	EDF	88.82	81.28	69.97	59.64	48.20	26.59	5.89	0.49	0.07	0.03
DLX 2	part.	91.46	88.43	84.93	79.81	73.18	64.45	53.36	38.95	20.97	6.26
	global 1	99.75	99.38	98.61	96.42	91.81	82.91	69.56	51.71	29.18	10.03
	global 2	99.74	99.33	98.49	96.17	91.38	82.54	69.32	51.74	29.24	10.07
Exp. 3a	EDF	99.84	99.30	97.41	92.72	82.57	66.05	45.26	24.15	6.77	0.05
DLX 5	part.	99.84	99.45	98.37	95.92	90.66	81.35	68.38	52.80	35.42	14.36
	global 1	100	100	100	99.93	98.68	90.87	75.11	56.73	38.59	17.53
	global 2	100	100	100	99.92	98.55	90.56	74.94	56.68	38.55	17.54

Table E.3: *Measured acceptance ratio (in %) for Experiment 3a (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3b	EDF	92.15	46.66	23.43	13.55	7.94	3.50	0.75	0.23	0.12	0.07
DLX 1	part.	92.49	92.09	91.01	88.54	82.24	69.61	52.44	35.60	19.09	5.94
	global 1	99.77	99.75	99.69	99.52	98.89	97.24	93.70	84.05	56.53	22.62
	global 2	100	100	100	100	100	100	99.86	95.77	70.32	28.21
Exp. 3b	EDF	95.53	93.45	89.86	62.66	26.40	9.47	1.78	0.36	0.11	0.06
DLX 1.5	part.	95.80	95.41	94.67	93.45	91.02	82.69	68.27	48.81	26.83	7.99
	global 1	100	100	100	100	100	100	99.95	98.57	78.78	33.61
	global 2	100	100	100	100	100	100	99.99	99.62	84.60	36.04
Exp. 3b	EDF	99.63	98.53	95.56	92.23	84.83	51.52	11.78	0.88	0.12	0.05
DLX 2	part.	99.64	99.16	98.57	97.41	95.45	92.24	85.02	67.88	38.37	10.82
	global 1	100	100	100	100	100	100	100	99.99	91.87	39.33
	global 2	100	100	100	100	100	100	100	100	95.16	41.85
Exp. 3b	EDF	100	100	100	99.99	99.94	99.45	96.66	82.78	38.30	0.19
DLX 5	part.	100	100	100	100	99.97	99.83	99.04	95.03	80.26	35.52
	global 1	100	100	100	100	100	100	100	100	100	74.38
	global 2	100	100	100	100	100	100	100	100	100	76.19

Table E.4: *Measured acceptance ratio (in %) for Experiment 3b (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3b	EDF	84.37	42.61	21.25	12.47	7.26	3.23	0.69	0.21	0.10	0.07
DLX 1	part.	85.69	85.14	83.63	80.67	74.23	62.82	46.66	31.42	16.71	5.09
	global 1	98.53	98.38	98.07	97.33	95.56	91.85	84.44	67.74	41.17	15.74
	global 2	100	100	100	100	100	99.99	98.54	83.67	53.15	19.41
Exp. 3b	EDF	90.38	86.56	81.24	55.29	23.44	8.49	1.57	0.33	0.10	0.05
DLX 1.5	part.	91.50	90.76	89.44	87.19	83.46	74.58	60.25	42.53	23.16	6.84
	global 1	100	100	100	100	100	99.99	99.33	88.58	58.33	23.10
	global 2	100	100	100	100	100	100	99.97	94.61	64.10	24.65
Exp. 3b	EDF	98.41	95.81	89.99	83.92	74.18	43.76	9.92	0.74	0.11	0.04
DLX 2	part.	98.56	97.46	96.11	93.76	90.05	84.37	74.79	57.47	31.74	9.00
	global 1	100	100	100	100	100	100	100	98.67	68.75	26.54
	global 2	100	100	100	100	100	100	100	99.61	72.15	27.94
Exp. 3b	EDF	100	100	99.98	99.88	99.25	96.46	87.30	63.59	24.70	0.14
DLX 5	part.	100	100	99.98	99.93	99.68	98.64	95.03	84.69	63.51	25.78
	global 1	100	100	100	100	100	100	100	100	93.74	44.03
	global 2	100	100	100	100	100	100	100	100	92.99	44.50

Table E.5: *Measured acceptance ratio (in %) for Experiment 3b (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3b	EDF	64.29	31.85	15.97	9.41	5.58	2.48	0.52	0.17	0.08	0.05
DLX 1	part.	70.31	69.27	67.09	63.47	57.34	47.87	35.48	23.55	12.43	3.79
	global 1	88.02	87.08	85.36	82.39	77.54	70.12	57.22	39.62	23.04	8.82
	global 2	100	99.99	99.96	99.66	97.23	88.17	71.14	50.30	29.78	10.62
Exp. 3b	EDF	73.68	66.54	58.60	38.29	16.38	6.00	1.15	0.24	0.07	0.04
DLX 1.5	part.	79.69	78.00	75.28	71.18	65.50	56.53	44.33	30.58	16.43	4.93
	global 1	99.85	99.79	99.63	99.19	97.22	88.10	70.39	49.86	30.59	12.21
	global 2	100	100	100	99.98	99.38	92.77	76.65	55.92	33.42	12.86
Exp. 3b	EDF	88.88	81.46	69.95	59.67	47.95	26.52	5.91	0.49	0.07	0.03
DLX 2	part.	91.50	88.60	84.92	79.85	72.99	64.37	53.40	39.02	21.10	6.24
	global 1	100	100	100	99.99	99.72	94.49	78.11	58.75	34.34	13.45
	global 2	100	100	100	100	99.86	94.68	78.54	59.49	35.24	13.83
Exp. 3b	EDF	99.82	99.26	97.44	92.61	82.38	66.13	45.23	24.21	6.85	0.05
DLX 5	part.	99.83	99.40	98.37	95.86	90.52	81.39	68.39	52.80	35.48	14.47
	global 1	100	100	100	100	100	95.84	75.11	55.93	39.17	19.41
	global 2	100	100	100	100	100	95.34	75.19	55.92	39.10	19.49

Table E.6: *Measured acceptance ratio (in %) for Experiment 3b (50% aperiodic job utilization).*

E.2 Quickness

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	partitioned	0.99	0.99	0.99	0.98	0.98	0.96	0.93	0.91	0.90	0.91
	global 1	0.91	0.91	0.89	0.86	0.79	0.66	0.53	0.43	0.36	0.29
	global 2	0.91	0.90	0.89	0.86	0.78	0.66	0.51	0.41	0.34	0.28
DLX 1.5	partitioned	0.97	0.97	0.97	0.96	0.93	0.88	0.81	0.78	0.76	0.78
	global 1	0.97	0.96	0.96	0.95	0.91	0.85	0.76	0.70	0.65	0.66
	global 2	0.96	0.96	0.96	0.95	0.91	0.85	0.75	0.70	0.66	0.66
DLX 2	partitioned	0.94	0.95	0.95	0.94	0.92	0.86	0.76	0.68	0.65	0.70
	global 1	0.94	0.95	0.95	0.94	0.92	0.86	0.75	0.66	0.59	0.62
	global 2	0.94	0.95	0.95	0.94	0.92	0.85	0.74	0.64	0.58	0.62
DLX 5	partitioned	0.98	0.98	0.98	0.97	0.96	0.93	0.86	0.77	0.61	0.42
	global 1	0.98	0.98	0.98	0.97	0.96	0.93	0.86	0.77	0.59	0.35
	global 2	0.98	0.98	0.98	0.97	0.96	0.93	0.86	0.76	0.57	0.34

Table E.7: *Linux Experiment 3a: average quickness, 4 cores,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	partitioned	0.98	0.98	0.98	0.98	0.97	0.95	0.93	0.91	0.89	0.89
	global 1	0.82	0.82	0.79	0.76	0.68	0.58	0.47	0.40	0.33	0.27
	global 2	0.82	0.81	0.79	0.75	0.68	0.57	0.46	0.38	0.31	0.27
DLX 1.5	partitioned	0.95	0.95	0.94	0.93	0.90	0.86	0.80	0.76	0.75	0.76
	global 1	0.93	0.93	0.92	0.91	0.86	0.80	0.71	0.65	0.62	0.62
	global 2	0.92	0.92	0.92	0.90	0.86	0.80	0.71	0.65	0.62	0.63
DLX 2	partitioned	0.88	0.90	0.90	0.90	0.88	0.82	0.73	0.66	0.62	0.67
	global 1	0.88	0.90	0.90	0.89	0.87	0.80	0.69	0.59	0.53	0.58
	global 2	0.88	0.90	0.89	0.88	0.86	0.79	0.67	0.58	0.53	0.58
DLX 5	partitioned	0.95	0.96	0.95	0.94	0.91	0.86	0.77	0.66	0.54	0.38
	global 1	0.95	0.96	0.95	0.94	0.91	0.86	0.76	0.62	0.42	0.28
	global 2	0.95	0.96	0.95	0.94	0.91	0.85	0.75	0.59	0.41	0.28

Table E.8: *Linux Experiment 3a: average quickness, 4 cores,* $U_{aperiodic} = 20\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	partitioned	0.96	0.96	0.96	0.95	0.95	0.93	0.91	0.89	0.87	0.88
	global 1	0.61	0.60	0.58	0.54	0.49	0.43	0.38	0.34	0.30	0.26
	global 2	0.60	0.59	0.56	0.52	0.47	0.41	0.35	0.32	0.28	0.26
DLX 1.5	partitioned	0.87	0.87	0.87	0.86	0.83	0.80	0.75	0.72	0.71	0.73
	global 1	0.81	0.80	0.78	0.75	0.71	0.65	0.58	0.55	0.54	0.57
	global 2	0.79	0.78	0.76	0.74	0.69	0.64	0.59	0.56	0.55	0.57
DLX 2	partitioned	0.74	0.78	0.77	0.77	0.76	0.72	0.65	0.59	0.57	0.63
	global 1	0.73	0.75	0.73	0.70	0.65	0.57	0.49	0.44	0.44	0.50
	global 2	0.71	0.73	0.71	0.67	0.63	0.56	0.49	0.43	0.44	0.50
DLX 5	partitioned	0.83	0.85	0.81	0.75	0.67	0.58	0.49	0.42	0.37	0.31
	global 1	0.83	0.85	0.80	0.73	0.61	0.42	0.28	0.21	0.18	0.20
	global 2	0.83	0.85	0.80	0.72	0.58	0.40	0.27	0.20	0.18	0.19

Table E.9: *Linux Experiment 3a: average quickness, 4 cores,* $U_{aperiodic} = 50\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	partitioned	0.99	0.99	0.99	0.98	0.98	0.96	0.94	0.92	0.90	0.91
	global 1	0.91	0.91	0.89	0.86	0.79	0.64	0.45	0.28	0.17	0.10
	global 2	0.91	0.90	0.89	0.86	0.78	0.62	0.42	0.23	0.12	0.06
DLX 1.5	partitioned	0.97	0.97	0.97	0.96	0.93	0.88	0.82	0.78	0.76	0.78
	global 1	0.97	0.96	0.96	0.95	0.91	0.86	0.79	0.67	0.51	0.48
	global 2	0.96	0.96	0.96	0.95	0.91	0.84	0.75	0.65	0.55	0.53
DLX 2	partitioned	0.94	0.95	0.95	0.94	0.92	0.86	0.76	0.68	0.65	0.69
	global 1	0.94	0.95	0.95	0.94	0.92	0.86	0.77	0.69	0.51	0.47
	global 2	0.94	0.95	0.95	0.94	0.92	0.85	0.74	0.63	0.50	0.49
DLX 5	partitioned	0.98	0.98	0.98	0.97	0.96	0.93	0.86	0.77	0.62	0.42
	global 1	0.98	0.98	0.98	0.97	0.96	0.93	0.86	0.77	0.64	0.25
	global 2	0.98	0.98	0.98	0.97	0.96	0.93	0.86	0.76	0.58	0.21

Table E.10: *Linux Experiment 3b: average quickness, 32 cores,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	partitioned	0.98	0.98	0.98	0.98	0.97	0.95	0.93	0.91	0.89	0.90
	global 1	0.83	0.82	0.80	0.76	0.69	0.55	0.37	0.23	0.16	0.09
	global 2	0.81	0.80	0.78	0.74	0.65	0.49	0.28	0.14	0.08	0.05
DLX 1.5	partitioned	0.95	0.95	0.94	0.93	0.90	0.86	0.80	0.77	0.75	0.77
	global 1	0.93	0.93	0.92	0.91	0.87	0.82	0.69	0.50	0.40	0.41
	global 2	0.92	0.92	0.92	0.90	0.86	0.79	0.67	0.54	0.46	0.47
DLX 2	partitioned	0.88	0.90	0.90	0.90	0.88	0.82	0.73	0.65	0.62	0.67
	global 1	0.88	0.90	0.90	0.89	0.87	0.82	0.73	0.56	0.39	0.39
	global 2	0.88	0.90	0.90	0.88	0.86	0.79	0.67	0.51	0.37	0.41
DLX 5	partitioned	0.95	0.96	0.95	0.94	0.91	0.86	0.77	0.66	0.54	0.38
	global 1	0.95	0.96	0.95	0.94	0.91	0.86	0.77	0.66	0.27	0.16
	global 2	0.95	0.96	0.95	0.94	0.91	0.85	0.75	0.59	0.23	0.14

Table E.11: *Linux Experiment 3b: average quickness, 32 cores,* $U_{aperiodic} = 20\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	partitioned	0.96	0.96	0.96	0.95	0.95	0.93	0.91	0.89	0.87	0.88
	global 1	0.66	0.65	0.62	0.58	0.50	0.39	0.27	0.21	0.16	0.08
	global 2	0.52	0.49	0.44	0.35	0.23	0.14	0.09	0.07	0.05	0.04
DLX 1.5	partitioned	0.87	0.87	0.87	0.86	0.84	0.80	0.75	0.72	0.71	0.73
	global 1	0.80	0.80	0.78	0.75	0.67	0.49	0.35	0.30	0.27	0.29
	global 2	0.78	0.76	0.74	0.68	0.57	0.42	0.34	0.32	0.32	0.34
DLX 2	partitioned	0.75	0.78	0.77	0.77	0.76	0.72	0.65	0.59	0.57	0.63
	global 1	0.73	0.76	0.75	0.73	0.67	0.47	0.31	0.27	0.26	0.26
	global 2	0.71	0.73	0.70	0.65	0.54	0.34	0.23	0.21	0.23	0.28
DLX 5	partitioned	0.83	0.85	0.81	0.75	0.67	0.58	0.49	0.42	0.37	0.31
	global 1	0.83	0.85	0.81	0.75	0.66	0.30	0.12	0.08	0.06	0.09
	global 2	0.83	0.85	0.80	0.72	0.57	0.22	0.09	0.07	0.06	0.08

Table E.12: *Linux Experiment 3b: average quickness, 32 cores,* $U_{aperiodic} = 50\%$.

E.3 Quickness Improvement with SDL

Quickness Improvement	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3a DLX 1	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	global 1	0.0	0.0	0.0	0.0	0.0	0.0	-0.1	0.1	-0.1	-0.2
	global 2	0.0	0.0	0.0	-0.1	0.0	0.1	-0.1	-0.1	0.1	-0.1
Exp. 3a DLX 1.5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	1.0	2.6	4.5	5.7	6.1	6.0	6.4	6.6	7.6
	global 1	0.0	1.0	2.5	4.3	5.5	5.8	5.8	5.8	5.5	6.0
	global 2	0.0	1.0	2.6	4.4	5.6	5.9	5.5	5.8	5.4	6.0
Exp. 3a DLX 2	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-1.7	2.8	7.1	13.3	19.3	24.7	27.4	28.6	27.4	26.1
	global 1	-1.7	2.7	7.0	13.2	19.1	24.5	27.2	27.9	24.6	21.6
	global 2	-1.7	2.7	7.1	13.2	19.2	24.4	26.7	27.4	24.8	21.9
Exp. 3a DLX 5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-1.1	2.8	6.7	11.8	19.1	30.8	52.8	96.5	166.2	160.5
	global 1	-1.1	2.8	6.7	11.8	19.1	30.8	52.9	96.9	163.8	137.0
	global 2	-1.1	2.8	6.7	11.8	19.1	30.8	52.9	96.9	160.5	135.6

Table E.13: *Experiment 3a: quickness improvement (in %), 4 cores, 10% aperiodic job utilization.*

Quickness Improvement	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3a DLX 1	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	global 2	0.0	0.0	-0.1	0.0	0.0	-0.1	0.1	-0.1	-0.1	0.0
	v4	0.0	0.0	0.0	0.0	0.1	0.0	-0.1	-0.4	-0.1	-0.2
Exp. 3a DLX 1.5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	1.2	2.7	4.8	6.0	6.2	6.4	6.4	6.3	7.3
	global 2	0.0	1.1	2.6	4.5	5.7	5.8	5.9	5.6	5.1	5.4
	v4	0.0	1.2	2.8	4.7	5.8	5.9	5.9	5.3	5.0	5.3
Exp. 3a DLX 2	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-3.1	2.9	7.8	13.8	20.1	24.5	27.0	28.6	26.7	24.6
	global 2	-3.1	2.8	7.6	13.6	19.8	24.4	26.5	26.6	21.9	18.8
	v4	-3.1	2.7	7.6	13.7	19.9	23.9	25.7	26.0	22.2	19.0
Exp. 3a DLX 5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-2.3	2.8	7.2	12.9	21.0	33.9	57.1	98.0	155.4	148.8
	global 2	-2.3	2.8	7.2	12.9	21.0	33.9	57.5	98.5	135.2	108.3
	v4	-2.3	2.8	7.2	12.9	21.0	33.9	57.3	97.1	132.6	107.3

Table E.14: *Experiment 3a: quickness improvement (in %), 4 cores, 20% aperiodic job utilization.*

Quickness Improvement	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3a DLX 1	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	global 1	0.1	-0.1	0.2	-0.4	-0.1	-0.1	0.0	0.0	-0.2	0.2
	global 2	-0.1	0.1	-0.2	0.1	0.0	-0.5	-0.3	-0.2	-0.1	0.1
Exp. 3a DLX 1.5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	1.6	3.5	5.4	6.2	6.3	6.4	6.2	6.0	6.8
	global 1	0.0	1.7	3.6	5.4	6.0	5.8	5.5	4.9	4.2	4.5
	global 2	0.0	1.8	3.8	5.5	5.9	5.6	5.3	4.9	4.3	4.4
Exp. 3a DLX 2	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-5.5	3.5	9.1	14.9	20.9	24.4	26.3	26.5	24.2	22.5
	global 1	-5.4	3.5	9.2	15.2	21.1	23.0	22.4	19.7	16.5	14.8
	global 2	-5.5	3.4	9.3	15.0	20.4	21.9	21.7	19.6	16.8	14.7
Exp. 3a DLX 5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-6.3	3.0	9.1	17.4	28.7	43.3	64.2	92.8	124.5	120.8
	global 1	-6.3	3.0	9.1	17.6	30.2	45.8	59.1	64.3	63.1	69.0
	global 2	-6.3	2.9	9.1	17.5	30.0	44.9	57.4	63.1	62.8	67.5

Table E.15: *Experiment 3a: quickness improvement (in %), 4 cores, 50% aperiodic job utilization.*

Quickness Improvement	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3b DLX 1	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	global 2	0.0	0.0	0.0	0.0	0.0	0.0	-0.1	0.2	-0.1	-0.6
	v4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.2	-0.4	-0.2
Exp. 3b DLX 1.5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	1.0	2.5	4.6	5.6	6.0	6.3	6.4	6.4	7.4
	global 2	0.0	1.0	2.4	4.3	5.2	5.1	5.5	6.8	4.8	4.0
	v4	0.0	1.0	2.5	4.5	5.5	5.8	5.8	5.5	4.4	4.2
Exp. 3b DLX 2	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-1.7	2.8	7.2	13.2	19.7	24.7	27.2	28.7	27.3	25.7
	global 2	-1.7	2.7	7.1	12.9	19.0	23.6	26.5	30.0	21.1	13.7
	v4	-1.7	2.7	7.2	13.1	19.6	24.4	26.6	27.0	22.3	15.4
Exp. 3b DLX 5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-1.1	2.8	6.7	11.8	19.1	30.8	52.9	96.3	167.2	164.0
	global 2	-1.1	2.8	6.7	11.8	19.1	30.8	52.9	96.1	168.8	96.2
	v4	-1.1	2.8	6.7	11.8	19.1	30.8	53.0	96.5	161.8	82.3

Table E.16: *Experiment 3b: quickness improvement (in %), 32 cores, 10% aperiodic job utilization.*

Quickness Improvement	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3b DLX 1	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	global 2	0.0	0.0	0.0	0.0	-0.1	-0.1	0.1	-0.1	0.0	0.1
	v4	0.0	0.0	-0.1	0.0	-0.1	0.0	0.1	0.3	1.1	-1.0
Exp. 3b DLX 1.5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	1.2	2.8	4.8	5.9	6.1	6.3	6.4	6.3	7.3
	global 2	0.0	1.1	2.5	4.3	5.0	5.0	6.7	6.0	3.7	2.9
	v4	0.0	1.2	2.7	4.7	5.7	5.8	5.5	5.0	3.5	3.2
Exp. 3b DLX 2	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-3.1	2.9	7.7	13.8	20.0	24.8	27.2	28.2	26.8	25.0
	global 2	-3.1	2.8	7.4	13.1	18.7	23.7	28.6	27.3	13.4	9.6
	v4	-3.1	2.8	7.5	13.6	19.8	24.3	26.0	24.1	15.1	10.9
Exp. 3b DLX 5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-2.3	2.8	7.2	12.9	21.0	33.9	57.2	98.3	156.4	149.4
	global 2	-2.3	2.8	7.2	12.9	21.0	33.9	57.1	98.1	97.6	58.8
	v4	-2.3	2.8	7.2	12.9	21.0	33.9	57.5	97.4	86.4	45.3

Table E.17: *Experiment 3b: quickness improvement (in %), 32 cores, 20% aperiodic job utilization.*

Quickness Improvement	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 3b DLX 1	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	global 1	0.0	0.0	0.0	0.1	0.0	0.2	1.3	-0.1	0.0	-0.2
	global 2	0.0	0.1	-0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0
Exp. 3b DLX 1.5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	0.0	1.6	3.4	5.4	6.3	6.3	6.4	6.4	6.1	6.9
	global 1	0.0	1.5	3.0	4.6	6.5	7.6	4.7	3.1	2.3	1.6
	global 2	0.0	1.8	3.6	5.3	5.7	4.5	3.5	2.8	2.4	1.7
Exp. 3b DLX 2	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-5.5	3.4	9.0	15.1	20.8	24.6	26.5	26.7	24.6	22.3
	global 1	-5.1	3.6	8.9	15.2	23.9	24.1	12.6	9.1	7.3	4.8
	global 2	-5.5	3.4	8.8	15.1	19.7	17.3	11.0	8.4	7.7	5.5
Exp. 3b DLX 5	EDF	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	part.	-6.4	3.0	9.1	17.4	28.6	43.3	64.3	93.0	124.0	120.0
	global 1	-6.4	3.0	9.1	17.3	28.9	46.6	34.2	26.4	23.9	25.6
	global 2	-6.4	2.9	9.0	17.4	30.0	38.8	28.7	24.8	23.8	18.8

Table E.18: *Experiment 3b: quickness improvement (in %), 32 cores, 50% aperiodic job utilization.*

Appendix F

Detailed Results of Linux Experiment 4

This appendix lists the results of the effectiveness measurements we performed on Elwetritsch, the high performance cluster of the University of Kaiserslautern: **Experiment 4** - This experiment runs on a simulated quadcore system. Three different arrival settings for the aperiodic jobs have been analyzed to simulate a gradual shift from dedicated handling of aperiodic jobs, to handling aperiodic jobs on all cores.
The tables list the results for three distinct scenarios which differ in the utilization created by the arriving aperiodic jobs: 10%, 20%, and 50%. The experiment has been conducted for offline utilization of 50%; each result being based on 1000 simulated jobs sets. Furthermore, each table lists the results for DLX factors of 1.5, 2, and 5, 10, 15, 20. The following scheduling algorithms have been applied:

- EDF with background processing of aperiodic jobs
- partitioned slot shifting
- global spare-capacity-based slot shifting (global algorithm 1)
- global negotiation-based slot shifting (global algorithm 2).

F.1 Acceptance Ratio

Algorithm	DLX Factor					
	1.5	2	5	10	15	20
EDF with backgr. proc.	9.50	51.39	99.47	100.00	100.00	100.00
partitioned slot shifting	82.65	92.35	99.92	100.00	100.00	100.00
global slot shifting 1	99.27	99.94	100.00	100.00	100.00	100.00
global slot shifting 2	99.28	99.95	100.00	100.00	100.00	100.00
partitioned slot shifting (SDL)	82.65	92.32	99.86	100.00	100.00	100.00
global slot shifting 1 (SDL)	99.27	99.94	100.00	100.00	100.00	100.00
globalslot shifting 2 (SDL)	99.31	99.94	100.00	100.00	100.00	100.00

Table F.1: *Measured acceptance ratio (in %) for Experiment 4 (10% aperiodic task utilization).*

Algorithm	DLX Factor					
	1.5	2	5	10	15	20
EDF with backgr. proc.	8.43	43.73	96.42	99.96	100.00	100.00
partitioned slot shifting	74.48	84.36	98.86	99.99	100.00	100.00
global slot shifting 1	96.72	99.23	100.00	100.00	100.00	100.00
global slot shifting 2	96.76	99.19	100.00	100.00	100.00	100.00
partitioned slot shifting (SDL)	74.48	84.30	98.61	99.98	100.00	100.00
global slot shifting 1 (SDL)	96.71	99.22	100.00	100.00	100.00	100.00
globalslot shifting 2 (SDL)	96.74	99.17	100.00	100.00	100.00	100.00

Table F.2: *Measured acceptance ratio (in %) for Experiment 4 (20% aperiodic task utilization).*

Algorithm	DLX Factor					
	1.5	2	5	10	15	20
EDF with backgr. proc.	6.09	26.46	66.12	79.99	83.99	85.85
partitioned slot shifting	56.68	64.46	81.91	88.81	90.76	91.71
global slot shifting 1	77.24	83.06	91.12	93.13	93.38	93.52
globalslot shifting 2	77.09	82.63	90.82	92.97	93.30	93.47
partitioned slot shifting (SDL)	56.68	64.42	81.40	87.98	89.70	90.33
global slot shifting 1 (SDL)	77.26	83.05	90.99	92.79	92.74	92.56
global slot shifting 2 (SDL)	77.10	82.60	90.69	92.62	92.70	92.54

Table F.3: *Measured acceptance ratio (in %) for Experiment 4 (50% aperiodic task utilization).*

F.2 Quickness

Algorithm	DLX Factor					
	1.5	2	5	10	15	20
EDF with backgr. proc.	0.48	0.26	0.71	0.87	0.91	0.93
partitioned slot shifting	0.75	0.69	0.71	0.87	0.91	0.93
global slot shifting 1	0.61	0.69	0.71	0.87	0.91	0.93
global slot shifting 2	0.61	0.69	0.71	0.87	0.91	0.93
partitioned slot shifting (SDL)	0.88	0.86	0.93	0.97	0.98	0.98
global slot shifting 1 (SDL)	0.85	0.86	0.93	0.97	0.98	0.98
globalslot shifting 2 (SDL)	0.84	0.85	0.93	0.97	0.98	0.98

Table F.4: *Measured quickness for Experiment 4 (10% aperiodic task utilization).*

Algorithm	DLX Factor					
	1.5	2	5	10	15	20
EDF with backgr. proc.	0.50	0.27	0.65	0.82	0.88	0.91
partitioned slot shifting	0.81	0.66	0.64	0.82	0.88	0.91
global slot shifting 1	0.76	0.64	0.64	0.82	0.88	0.91
global slot shifting 2	0.75	0.63	0.64	0.82	0.88	0.91
partitioned slot shifting (SDL)	0.86	0.82	0.86	0.92	0.95	0.96
global slot shifting 1 (SDL)	0.80	0.80	0.86	0.92	0.95	0.96
globalslot shifting 2 (SDL)	0.80	0.79	0.85	0.92	0.95	0.96

Table F.5: *Measured quickness for Experiment 4 (20% aperiodic task utilization).*

Algorithm	DLX Factor					
	1.5	2	5	10	15	20
EDF with backgr. proc.	0.48	0.26	0.45	0.47	0.45	0.44
partitioned slot shifting	0.75	0.58	0.40	0.43	0.42	0.41
global slot shifting 1	0.61	0.47	0.29	0.28	0.28	0.30
global slot shifting 2	0.61	0.46	0.28	0.27	0.28	0.30
partitioned slot shifting (SDL)	0.80	0.72	0.58	0.53	0.50	0.48
global slot shifting 1 (SDL)	0.65	0.58	0.43	0.38	0.36	0.36
globalslot shifting 2 (SDL)	0.64	0.56	0.41	0.36	0.35	0.36

Table F.6: *Measured quickness for Experiment 4 (50% aperiodic task utilization).*

Appendix G

Detailed Results of Linux Experiment 5

This appendix lists the results of the effectiveness measurements we performed on Elwetritsch, the high performance cluster of the University of Kaiserslautern: **Experiment 5** - This experiment runs on a simulated 4-core system. Three different arrival settings for the aperiodic jobs have been analyzed to simulate a gradual shift from dedicated handling of aperiodic jobs, to handling aperiodic jobs on all cores.
The tables list the results for three distinct scenarios which differ in the utilization created by the arriving aperiodic jobs: 10%, 20%, and 50%. The experiment has been conducted for offline utilizations between 0% and 90% in steps of 10%; each result being based on 1000 simulated jobs sets. Furthermore, each table lists the results for DLX factors of 1, 1.5, 2, and 5. The following scheduling algorithms have been applied:

- EDF with background processing of aperiodic jobs
- partitioned slot shifting
- global spare-capacity-based slot shifting (global algorithm 1)
- global negotiation-based slot shifting (global algorithm 2).

G.1 Acceptance Ratio

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Exp. 5a	global 1	100	100	99.99	99.98	99.89	99.37	96.46	83.57	52.17	17.05
	global 2	100	100	99.99	99.97	99.84	99.22	96.32	83.82	52.50	17.17
Exp. 5b	global 1	100	100	99.99	99.96	99.85	99.25	96.30	82.98	51.69	16.91
	global 2	100	100	99.99	99.96	99.81	99.16	96.24	83.30	51.99	17.04
Exp. 5c	global 1	100	100	99.99	99.97	99.84	99.26	96.26	83.03	51.80	16.88
	global 2	100	100	99.99	99.97	99.83	99.21	96.16	83.23	52.06	16.96
Exp. 3	global 1	100	100	99.99	99.97	99.86	99.25	96.07	83.26	51.34	17.09
	global 2	100	100	99.99	99.97	99.85	99.19	96.03	83.43	51.61	17.20

Table G.1: *Experiment 5 with SDL: measured acceptance ratio in %, DLX factor 2,* $U_{aperiodic}$ *= 20%.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	99.66	99.66	99.56	99.36	98.76	94.76	82.63	63.79	38.56	13.87
	global 2	99.96	99.98	99.96	99.85	99.28	95.80	84.47	66.05	40.07	14.20
DLX 1.5	global 1	100	100	99.99	99.99	99.90	99.23	95.10	80.43	52.27	18.05
	global 2	100	100	99.99	99.99	99.91	99.31	95.46	81.11	52.77	18.18
DLX 2	global 1	100	100	100	100	100	99.96	99.51	94.24	67.87	23.21
	global 2	100	100	100	100	100	99.94	99.59	94.65	68.43	23.37
DLX 5	global 1	100	100	100	100	100	100	100	99.99	98.28	56.64
	global 2	100	100	100	100	100	100	100	99.98	97.99	56.82

Table G.2: *Experiment 5a: measured acceptance ratio in %, 10% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	98.78	98.60	98.45	97.66	95.80	89.22	74.04	54.54	30.54	10.36
	global 2	99.67	99.60	99.36	98.84	96.99	90.99	76.20	56.61	31.73	10.58
DLX 1.5	global 1	99.95	99.92	99.88	99.74	99.26	96.83	88.49	68.51	41.43	13.76
	global 2	99.95	99.92	99.86	99.69	99.18	96.94	88.90	69.19	41.82	13.82
DLX 2	global 1	100	100	100	99.98	99.89	99.38	96.46	83.58	52.16	17.05
	global 2	100	100	99.99	99.98	99.84	99.24	96.31	83.79	52.53	17.17
DLX 5	global 1	100	100	100	100	100	100	99.98	98.60	82.58	36.87
	global 2	100	100	100	100	100	100	99.95	98.08	81.67	36.97

Table G.3: *Experiment 5a: measured acceptance ratio in %, 20% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	91.72	90.42	88.67	85.07	78.46	68.07	52.65	35.70	19.74	6.29
	global 2	93.98	92.84	90.77	87.06	80.37	69.87	54.21	36.92	20.36	6.41
DLX 1.5	global 1	98.13	97.48	96.10	93.42	87.98	78.10	63.56	44.61	25.45	8.47
	global 2	98.01	97.19	95.58	92.63	87.17	77.75	63.57	44.92	25.59	8.51
DLX 2	global 1	99.85	99.65	99.00	97.24	92.70	84.13	70.42	52.23	29.54	10.16
	global 2	99.79	99.52	98.62	96.45	91.55	83.10	70.02	52.25	29.63	10.21
DLX 5	global 1	100	100	100	99.97	99.19	92.12	75.67	57.02	38.85	17.62
	global 2	100	100	100	99.90	98.48	90.76	75.10	56.87	38.80	17.64

Table G.4: *Experiment 5a: measured acceptance ratio in %, 50% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	99.72	99.72	99.67	99.48	98.65	94.72	82.98	64.83	38.58	13.17
	global 2	99.97	99.97	99.95	99.81	99.15	95.75	84.71	66.77	40.01	13.49
DLX 1.5	global 1	100	99.99	99.99	99.99	99.93	99.21	95.17	80.48	52.75	18.07
	global 2	100	99.99	99.99	99.98	99.94	99.29	95.48	81.09	53.23	18.18
DLX 2	global 1	100	100	100	100	100	99.95	99.50	93.91	66.58	22.95
	global 2	100	100	100	100	99.99	99.95	99.51	94.26	67.03	23.12
DLX 5	global 1	100	100	100	100	100	100	100	99.98	98.19	56.36
	global 2	100	100	100	100	100	100	100	99.98	97.98	56.53

Table G.5: *Experiment 5b: measured acceptance ratio in %, 10% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	98.75	98.49	98.25	97.57	95.54	88.68	73.58	53.48	30.80	10.45
	global 2	99.60	99.49	99.26	98.70	96.84	90.36	75.70	55.35	31.90	10.65
DLX 1.5	global 1	99.94	99.93	99.86	99.63	99.17	96.75	88.34	68.86	41.48	13.79
	global 2	99.94	99.93	99.85	99.61	99.14	96.83	88.68	69.43	41.87	13.88
DLX 2	global 1	100	100	100	99.97	99.86	99.25	96.30	82.97	51.68	16.92
	global 2	100	100	100	99.97	99.82	99.15	96.22	83.29	52.00	17.03
DLX 5	global 1	100	100	100	100	100	100	99.95	98.28	82.01	36.91
	global 2	100	100	100	100	100	100	99.92	97.95	81.42	36.96

Table G.6: *Experiment 5b: measured acceptance ratio in %, 20% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	91.02	89.89	88.03	84.35	78.13	67.63	52.59	35.05	19.62	6.22
	global 2	93.07	92.07	90.01	86.19	79.83	69.24	54.01	36.15	20.17	6.32
DLX 1.5	global 1	97.68	96.97	95.34	92.42	87.02	77.50	63.04	44.83	24.99	8.49
	global 2	97.63	96.82	95.06	91.91	86.49	77.27	63.17	45.13	25.16	8.54
DLX 2	global 1	99.71	99.41	98.50	96.48	91.75	83.10	69.88	51.96	29.61	10.13
	global 2	99.68	99.32	98.24	95.94	91.04	82.49	69.61	51.96	29.70	10.17
DLX 5	global 1	100	100	99.98	99.84	98.29	90.71	75.26	56.86	38.79	17.40
	global 2	100	100	99.98	99.71	97.72	89.85	74.86	56.77	38.78	17.42

Table G.7: *Experiment 5b: measured acceptance ratio in %, 50% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	99.89	99.87	99.84	99.60	98.92	95.01	83.39	63.93	38.29	13.38
	global 2	99.96	99.96	99.93	99.80	99.20	95.81	84.93	65.85	39.72	13.69
DLX 1.5	global 1	99.99	99.99	99.99	99.98	99.91	99.31	95.14	80.20	51.94	18.08
	global 2	99.99	99.99	99.99	99.97	99.91	99.35	95.43	80.76	52.37	18.18
DLX 2	global 1	100	100	100	100	100	99.96	99.57	94.15	67.62	22.99
	global 2	100	100	100	100	99.99	99.95	99.57	94.32	68.01	23.13
DLX 5	global 1	100	100	100	100	100	100	100	99.99	98.09	56.01
	global 2	100	100	100	100	100	100	100	99.99	97.94	56.12

Table G.8: *Experiment 5c: measured acceptance ratio in %, 10% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	99.28	99.12	98.90	98.13	95.86	89.15	73.88	53.53	30.38	10.26
	global 2	99.53	99.44	99.27	98.64	96.60	90.37	75.65	55.16	31.42	10.49
DLX 1.5	global 1	99.93	99.90	99.84	99.67	99.18	96.88	87.95	69.80	40.94	13.91
	global 2	99.93	99.90	99.82	99.65	99.15	96.95	88.29	70.32	41.22	13.97
DLX 2	global 1	100	100	100	99.97	99.85	99.26	96.27	83.02	51.78	16.88
	global 2	100	100	99.99	99.97	99.84	99.21	96.17	83.21	52.12	16.94
DLX 5	global 1	100	100	100	100	100	100	99.96	98.32	81.64	36.69
	global 2	100	100	100	100	100	100	99.95	98.17	81.27	36.76

Table G.9: *Experiment 5c: measured acceptance ratio in %, 20% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	92.08	91.01	88.89	85.04	78.38	67.66	52.48	35.69	19.58	6.26
	global 2	92.82	91.80	89.74	86.10	79.48	68.88	53.70	36.69	20.08	6.36
DLX 1.5	global 1	97.75	96.92	95.40	92.41	86.87	77.25	62.90	44.43	25.34	8.36
	global 2	97.64	96.74	95.13	92.05	86.47	77.03	62.95	44.65	25.47	8.39
DLX 2	global 1	99.76	99.46	98.67	96.49	91.61	83.01	69.65	51.64	29.16	9.92
	global 2	99.73	99.38	98.51	96.19	91.11	82.53	69.42	51.65	29.21	9.95
DLX 5	global 1	100	100	100	99.89	98.51	90.94	75.07	56.81	38.61	17.57
	global 2	100	100	99.99	99.84	98.18	90.37	74.84	56.75	38.56	17.57

Table G.10: *Experiment 5c: measured acceptance ratio in %, 50% aperiodic job utilization.*

G.2 Acceptance Ratio with SDL

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	99.63	99.65	99.56	99.44	98.74	94.74	82.63	63.83	38.54	13.87
	global 2	99.96	99.98	99.95	99.82	99.28	95.80	84.47	66.02	40.05	14.19
DLX 1.5	global 1	100	100	99.99	99.99	99.90	99.24	95.06	80.43	52.25	18.05
	global 2	100	100	99.99	99.99	99.92	99.30	95.47	81.09	52.72	18.17
DLX 2	global 1	100	100	100	100	100	99.96	99.52	94.23	67.84	23.21
	global 2	100	100	100	100	99.99	99.95	99.58	94.64	68.46	23.38
DLX 5	global 1	100	100	100	100	100	100	100	99.98	98.29	56.66
	global 2	100	100	100	100	100	100	100	99.98	97.94	56.79

Table G.11: *Experiment 5a with SDL: measured acceptance ratio in %, 10% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	98.79	98.58	98.43	97.64	95.80	89.27	73.99	54.57	30.56	10.35
	global 2	99.67	99.60	99.37	98.83	97.00	90.95	76.25	56.62	31.73	10.58
DLX 1.5	global 1	99.95	99.92	99.88	99.74	99.25	96.83	88.47	68.50	41.42	13.76
	global 2	99.95	99.92	99.87	99.70	99.18	96.91	88.91	69.10	41.82	13.82
DLX 2	global 1	100	100	99.99	99.98	99.89	99.37	96.46	83.57	52.17	17.05
	global 2	100	100	99.99	99.97	99.84	99.22	96.32	83.82	52.50	17.17
DLX 5	global 1	100	100	100	100	100	100	99.97	98.53	82.61	36.88
	global 2	100	100	100	100	100	100	99.94	98.00	81.66	36.99

Table G.12: *Experiment 5a with SDL: measured acceptance ratio in %, 20% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	91.77	90.44	88.65	85.09	78.46	68.04	52.63	35.71	19.76	6.29
	global 2	93.98	92.86	90.80	87.05	80.36	69.86	54.18	36.92	20.37	6.41
DLX 1.5	global 1	98.13	97.49	96.09	93.43	87.99	78.13	63.56	44.62	25.45	8.47
	global 2	98.01	97.20	95.59	92.65	87.21	77.79	63.61	44.90	25.59	8.52
DLX 2	global 1	99.84	99.61	98.95	97.20	92.72	84.13	70.42	52.25	29.54	10.16
	global 2	99.79	99.45	98.54	96.37	91.52	83.11	70.01	52.26	29.65	10.21
DLX 5	global 1	100	100	100	99.96	99.07	91.95	75.57	56.99	38.84	17.62
	global 2	100	100	100	99.88	98.41	90.69	75.05	56.85	38.79	17.62

Table G.13: *Experiment 5a with SDL: measured acceptance ratio in %, 50% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	99.72	99.73	99.67	99.45	98.62	94.73	82.98	64.86	38.59	13.16
	global 2	99.97	99.97	99.95	99.81	99.16	95.79	84.76	66.82	40.02	13.48
DLX 1.5	global 1	100	99.99	99.99	99.99	99.93	99.20	95.21	80.46	52.75	18.07
	global 2	100	99.99	99.99	99.98	99.94	99.29	95.49	81.05	53.18	18.21
DLX 2	global 1	100	100	100	100	99.99	99.95	99.51	93.90	66.57	22.95
	global 2	100	100	100	100	99.99	99.95	99.51	94.30	67.04	23.12
DLX 5	global 1	100	100	100	100	100	100	100	99.97	98.17	56.37
	global 2	100	100	100	100	100	100	100	99.97	97.90	56.59

Table G.14: *Experiment 5b with SDL: measured acceptance ratio in %, 10% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	98.78	98.50	98.28	97.62	95.54	88.70	73.60	53.47	30.79	10.45
	global 2	99.59	99.49	99.27	98.68	96.86	90.35	75.63	55.33	31.92	10.66
DLX 1.5	global 1	99.94	99.93	99.86	99.63	99.17	96.74	88.35	68.85	41.49	13.79
	global 2	99.93	99.93	99.85	99.62	99.14	96.82	88.71	69.39	41.87	13.89
DLX 2	global 1	100	100	99.99	99.96	99.85	99.25	96.30	82.98	51.69	16.91
	global 2	100	100	99.99	99.96	99.81	99.16	96.24	83.30	51.99	17.04
DLX 5	global 1	100	100	100	100	100	100	99.94	98.20	82.03	36.91
	global 2	100	100	100	100	100	100	99.92	97.86	81.39	36.94

Table G.15: *Experiment 5b with SDL: measured acceptance ratio in %, 20% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	91.22	90.32	88.28	84.67	78.34	67.73	52.64	35.07	19.61	6.22
	global 2	93.07	92.08	89.98	86.19	79.85	69.26	54.02	36.15	20.16	6.32
DLX 1.5	global 1	97.68	96.97	95.36	92.42	87.06	77.51	63.04	44.82	24.99	8.50
	global 2	97.62	96.81	95.08	91.90	86.49	77.23	63.20	45.10	25.15	8.54
DLX 2	global 1	99.71	99.34	98.42	96.41	91.73	83.12	69.86	51.96	29.62	10.13
	global 2	99.68	99.21	98.12	95.86	90.95	82.46	69.56	51.98	29.72	10.18
DLX 5	global 1	100	100	99.98	99.80	98.17	90.59	75.17	56.84	38.79	17.40
	global 2	100	100	99.97	99.68	97.66	89.75	74.78	56.74	38.75	17.41

Table G.16: *Experiment 5b with SDL: measured acceptance ratio in %, 50% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	99.89	99.88	99.84	99.64	98.95	95.07	83.42	63.98	38.31	13.39
	global 2	99.96	99.96	99.93	99.80	99.21	95.90	84.95	65.86	39.65	13.70
DLX 1.5	global 1	99.99	99.99	99.99	99.97	99.91	99.30	95.16	80.18	51.94	18.07
	global 2	99.99	99.99	99.99	99.97	99.91	99.35	95.41	80.76	52.37	18.18
DLX 2	global 1	100	100	100	100	100	99.95	99.57	94.17	67.61	23.01
	global 2	100	100	100	100	99.99	99.95	99.58	94.40	68.09	23.15
DLX 5	global 1	100	100	100	100	100	100	100	99.98	98.10	56.01
	global 2	100	100	100	100	100	100	100	99.98	97.95	56.19

Table G.17: *Experiment 5c with SDL: measured acceptance ratio in %, 10% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	99.27	99.13	98.87	98.15	95.88	89.13	73.86	53.54	30.38	10.26
	global 2	99.54	99.45	99.26	98.62	96.58	90.40	75.66	55.17	31.37	10.48
DLX 1.5	global 1	99.93	99.90	99.84	99.67	99.17	96.87	87.95	69.81	40.93	13.91
	global 2	99.93	99.90	99.84	99.65	99.14	96.96	88.33	70.29	41.23	13.98
DLX 2	global 1	100	100	99.99	99.97	99.84	99.26	96.26	83.03	51.80	16.88
	global 2	100	100	99.99	99.97	99.83	99.21	96.16	83.23	52.06	16.96
DLX 5	global 1	100	100	100	100	100	100	99.96	98.28	81.64	36.70
	global 2	100	100	100	100	100	100	99.95	98.07	81.18	36.71

Table G.18: *Experiment 5c with SDL: measured acceptance ratio in %, 20% aperiodic job utilization.*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	92.10	91.06	88.92	85.15	78.38	67.63	52.47	35.70	19.58	6.26
	global 2	92.82	91.81	89.75	86.10	79.48	68.89	53.69	36.70	20.10	6.36
DLX 1.5	global 1	97.75	96.92	95.42	92.40	86.87	77.25	62.89	44.42	25.34	8.36
	global 2	97.63	96.74	95.14	92.04	86.50	77.06	62.96	44.67	25.47	8.40
DLX 2	global 1	99.76	99.41	98.58	96.45	91.59	83.00	69.66	51.65	29.15	9.91
	global 2	99.73	99.32	98.43	96.12	91.11	82.50	69.42	51.66	29.19	9.96
DLX 5	global 1	100	100	99.99	99.87	98.43	90.83	74.97	56.78	38.61	17.57
	global 2	100	100	99.99	99.82	98.10	90.29	74.75	56.71	38.57	17.57

Table G.19: *Experiment 5c with SDL: measured acceptance ratio in %, 50% aperiodic job utilization.*

G.3 Average Number of Acceptance Tests

Average Number of Acceptance Tests		Utilization Created by Offline Jobs 0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Accepted Aperiodic Jobs											
Exp. 5a	global 1	1.19	1.23	1.28	1.34	1.42	1.50	1.60	1.69	1.77	1.77
	global 2	1.19	1.24	1.29	1.36	1.47	1.59	1.78	2.03	2.28	2.41
Exp. 5b	global 1	1.06	1.08	1.11	1.16	1.23	1.32	1.44	1.59	1.71	1.75
	global 2	1.06	1.09	1.12	1.18	1.26	1.39	1.59	1.88	2.18	2.35
Exp. 5c	global 1	1.02	1.04	1.05	1.09	1.13	1.21	1.33	1.51	1.67	1.73
	global 2	1.02	1.04	1.06	1.10	1.16	1.27	1.46	1.79	2.13	2.34
Finally Rejected Aperiodic Jobs											
Exp. 5a	global 1	–	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.97	3.96
	global 2	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
Exp. 5b	global 1	–	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.97	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
Exp. 5c	global 1	–	4.00	4.00	3.98	4.00	4.00	3.99	3.97	3.97	3.95
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
All Aperiodic Jobs											
Exp. 5a	global 1	1.19	1.23	1.28	1.34	1.42	1.52	1.68	2.07	2.82	3.58
	global 2	1.19	1.24	1.29	1.36	1.47	1.61	1.86	2.34	3.09	3.72
Exp. 5b	global 1	1.06	1.08	1.11	1.16	1.23	1.34	1.54	1.99	2.80	3.58
	global 2	1.06	1.09	1.12	1.18	1.27	1.42	1.68	2.23	3.05	3.72
Exp. 5c	global 1	1.02	1.04	1.05	1.09	1.14	1.23	1.43	1.93	2.77	3.57
	global 2	1.02	1.04	1.06	1.10	1.16	1.29	1.56	2.16	3.02	3.71

Table G.20: *Average number of acceptance tests per aperiodic job for Experiment 5 with SDL (20% aperiodic job utilization, DLX factor 2).*

Name		Utilization Created by Offline Jobs 0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.26	1.26	1.29	1.32	1.39	1.52	1.78	2.10	2.48	2.82
	global 2	1.27	1.28	1.32	1.37	1.49	1.75	2.20	2.73	3.30	3.77
DLX 1.5	global 1	1.17	1.18	1.21	1.24	1.30	1.41	1.63	2.08	2.78	3.57
	global 2	1.17	1.19	1.22	1.27	1.33	1.50	1.81	2.36	3.04	3.70
DLX 2	global 1	1.06	1.07	1.10	1.15	1.22	1.30	1.42	1.70	2.43	3.45
	global 2	1.06	1.08	1.11	1.16	1.24	1.34	1.50	1.94	2.71	3.61
DLX 5	global 1	1.00	1.00	1.00	1.01	1.04	1.10	1.22	1.38	1.62	2.72
	global 2	1.00	1.00	1.00	1.01	1.04	1.11	1.22	1.40	1.78	3.02

Table G.21: *Average number of acceptance tests per aperiodic job for Experiment 5a (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.99	3.99	3.98	3.97
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	–	–	–	4.00	4.00	3.99	3.99	3.98	3.97
	global 2	–	–	–	–	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	–	4.00	3.99	3.97
	global 2	–	–	–	–	–	–	–	4.00	4.00	4.00

Table G.22: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5a (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.25	1.26	1.28	1.31	1.37	1.44	1.52	1.59	1.66	1.70
	global 2	1.27	1.28	1.31	1.36	1.48	1.65	1.88	2.09	2.27	2.38
DLX 1.5	global 1	1.17	1.18	1.21	1.24	1.30	1.39	1.51	1.62	1.70	1.72
	global 2	1.17	1.19	1.22	1.27	1.33	1.48	1.71	1.99	2.21	2.35
DLX 2	global 1	1.06	1.07	1.10	1.15	1.22	1.30	1.41	1.56	1.69	1.73
	global 2	1.06	1.08	1.11	1.16	1.24	1.34	1.49	1.82	2.13	2.32
DLX 5	global 1	1.00	1.00	1.00	1.01	1.04	1.10	1.22	1.38	1.58	1.76
	global 2	1.00	1.00	1.00	1.01	1.04	1.11	1.22	1.40	1.74	2.28

Table G.23: *Average number of acceptance tests per accepted aperiodic job for Experiment 5a (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.16	1.17	1.19	1.23	1.30	1.46	1.74	2.06	2.47	2.83
	global 2	1.17	1.18	1.20	1.25	1.37	1.65	2.13	2.67	3.27	3.78
DLX 1.5	global 1	1.09	1.10	1.11	1.14	1.18	1.29	1.54	2.03	2.76	3.57
	global 2	1.09	1.10	1.13	1.16	1.20	1.36	1.71	2.28	3.01	3.70
DLX 2	global 1	1.01	1.02	1.03	1.06	1.10	1.17	1.29	1.61	2.42	3.46
	global 2	1.02	1.02	1.04	1.07	1.11	1.19	1.34	1.79	2.70	3.61
DLX 5	global 1	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.15	1.44	2.69
	global 2	1.00	1.00	1.00	1.00	1.00	1.01	1.06	1.19	1.59	2.97

Table G.24: *Average number of acceptance tests per aperiodic job for Experiment 5b (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	–	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.98	3.97
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	–	–	–	4.00	3.98	4.00	3.98	3.98	3.97
	global 2	–	–	4.00	–	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	–	4.00	3.99	3.96
	global 2	–	–	–	–	–	–	–	4.00	4.00	4.00

Table G.25: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5b (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.16	1.17	1.18	1.22	1.28	1.38	1.49	1.56	1.63	1.69
	global 2	1.17	1.18	1.20	1.24	1.35	1.55	1.81	2.03	2.21	2.38
DLX 1.5	global 1	1.09	1.10	1.11	1.14	1.18	1.27	1.41	1.57	1.67	1.73
	global 2	1.09	1.10	1.13	1.16	1.20	1.34	1.61	1.90	2.17	2.38
DLX 2	global 1	1.01	1.02	1.03	1.06	1.10	1.16	1.27	1.46	1.65	1.73
	global 2	1.02	1.02	1.04	1.07	1.11	1.19	1.33	1.66	2.08	2.33
DLX 5	global 1	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.15	1.39	1.70
	global 2	1.00	1.00	1.00	1.00	1.00	1.01	1.06	1.19	1.54	2.20

Table G.26: *Average number of acceptance tests per accepted aperiodic job for Experiment 5b (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.10	1.10	1.11	1.14	1.22	1.39	1.70	2.06	2.46	2.82
	global 2	1.11	1.12	1.13	1.18	1.30	1.58	2.09	2.66	3.27	3.77
DLX 1.5	global 1	1.05	1.06	1.07	1.08	1.12	1.22	1.48	2.00	2.76	3.56
	global 2	1.06	1.07	1.07	1.10	1.14	1.29	1.64	2.25	3.01	3.69
DLX 2	global 1	1.00	1.01	1.02	1.03	1.06	1.10	1.20	1.53	2.37	3.45
	global 2	1.00	1.01	1.02	1.03	1.06	1.12	1.26	1.70	2.63	3.60
DLX 5	global 1	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.07	1.32	2.67
	global 2	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.08	1.44	2.94

Table G.27: *Average number of acceptance tests per aperiodic job for Experiment 5c (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.98	3.97
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	–	4.00	4.00	4.00	4.00	4.00	3.98	3.97	3.97
	global 2	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	–	4.00	3.99	3.96
	global 2	–	–	–	–	–	–	–	4.00	4.00	4.00

Table G.28: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5c (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.09	1.10	1.11	1.14	1.20	1.31	1.45	1.54	1.61	1.68
	global 2	1.11	1.11	1.13	1.17	1.28	1.48	1.77	1.99	2.19	2.35
DLX 1.5	global 1	1.05	1.06	1.07	1.08	1.11	1.20	1.36	1.52	1.65	1.72
	global 2	1.06	1.07	1.07	1.10	1.14	1.27	1.53	1.85	2.14	2.34
DLX 2	global 1	1.00	1.01	1.02	1.03	1.06	1.10	1.19	1.38	1.61	1.72
	global 2	1.00	1.01	1.02	1.03	1.06	1.12	1.24	1.57	2.01	2.31
DLX 5	global 1	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.07	1.27	1.67
	global 2	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.08	1.39	2.14

Table G.29: *Average number of acceptance tests per accepted aperiodic job for Experiment 5c (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.42	1.43	1.46	1.50	1.58	1.73	1.97	2.26	2.59	2.87
	global 2	1.51	1.55	1.60	1.84	1.83	2.14	2.56	3.01	3.48	3.83
DLX 1.5	global 1	1.32	1.34	1.38	1.43	1.50	1.64	1.91	2.42	3.05	3.65
	global 2	1.37	1.42	1.47	1.53	1.66	1.85	2.24	2.76	3.30	3.79
DLX 2	global 1	1.19	1.22	1.28	1.34	1.43	1.53	1.69	2.08	2.82	3.58
	global 2	1.21	1.26	1.34	1.43	1.54	1.68	1.92	2.41	3.12	3.73
DLX 5	global 1	1.03	1.07	1.13	1.21	1.31	1.42	1.53	1.69	2.14	3.16
	global 2	1.04	1.08	1.15	1.25	1.38	1.51	1.67	1.92	2.53	3.42

Table G.30: *Average number of acceptance tests per aperiodic job for Experiment 5a (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.97	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	–	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.96
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	3.98	3.99	3.98	3.95
	global 2	–	–	–	–	–	–	4.00	4.00	4.00	4.00

Table G.31: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5a (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.40	1.41	1.43	1.47	1.51	1.57	1.61	1.65	1.68	1.70
	global 2	1.50	1.54	1.59	1.82	1.76	1.95	2.12	2.25	2.37	2.45
DLX 1.5	global 1	1.31	1.34	1.37	1.42	1.48	1.56	1.64	1.71	1.74	1.75
	global 2	1.37	1.41	1.47	1.52	1.64	1.78	2.02	2.20	2.34	2.46
DLX 2	global 1	1.19	1.22	1.28	1.34	1.42	1.51	1.61	1.70	1.77	1.77
	global 2	1.21	1.26	1.34	1.43	1.53	1.66	1.84	2.10	2.34	2.45
DLX 5	global 1	1.03	1.07	1.13	1.21	1.31	1.42	1.53	1.65	1.74	1.80
	global 2	1.04	1.08	1.15	1.25	1.38	1.51	1.67	1.88	2.20	2.43

Table G.32: *Average number of acceptance tests per accepted aperiodic job for Experiment 5a (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.31	1.32	1.35	1.39	1.48	1.65	1.93	2.25	2.58	2.86
	global 2	1.34	1.36	1.40	1.52	1.62	1.96	2.43	2.96	3.44	3.83
DLX 1.5	global 1	1.19	1.20	1.23	1.27	1.34	1.50	1.82	2.36	3.02	3.65
	global 2	1.20	1.23	1.26	1.32	1.44	1.64	2.05	2.63	3.26	3.77
DLX 2	global 1	1.06	1.07	1.11	1.16	1.23	1.34	1.54	1.99	2.80	3.58
	global 2	1.06	1.08	1.12	1.19	1.27	1.43	1.69	2.25	3.06	3.72
DLX 5	global 1	1.00	1.00	1.00	1.01	1.04	1.10	1.22	1.45	2.03	3.14
	global 2	1.00	1.00	1.00	1.01	1.05	1.12	1.30	1.62	2.33	3.38

Table G.33: *Average number of acceptance tests per aperiodic job for Experiment 5b (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.99	3.97	3.96	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	–	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	4.00	3.98	3.99	3.98	3.95
	global 2	–	–	–	–	–	4.00	4.00	4.00	4.00	4.00

Table G.34: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5b (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.29	1.30	1.32	1.36	1.41	1.48	1.54	1.60	1.65	1.69
	global 2	1.33	1.35	1.38	1.49	1.54	1.75	1.94	2.14	2.27	2.37
DLX 1.5	global 1	1.19	1.20	1.23	1.26	1.32	1.41	1.53	1.63	1.70	1.73
	global 2	1.20	1.23	1.26	1.31	1.41	1.56	1.81	2.04	2.24	2.39
DLX 2	global 1	1.06	1.07	1.11	1.15	1.23	1.32	1.44	1.59	1.71	1.75
	global 2	1.06	1.08	1.12	1.19	1.27	1.41	1.60	1.90	2.20	2.38
DLX 5	global 1	1.00	1.00	1.00	1.01	1.04	1.10	1.22	1.40	1.61	1.76
	global 2	1.00	1.00	1.00	1.01	1.05	1.12	1.30	1.57	1.95	2.33

Table G.35: *Average number of acceptance tests per accepted aperiodic job for Experiment 5b (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.19	1.20	1.22	1.26	1.37	1.57	1.89	2.23	2.58	2.86
	global 2	1.27	1.30	1.34	1.41	1.59	1.92	2.45	2.96	3.49	3.83
DLX 1.5	global 1	1.11	1.12	1.14	1.18	1.24	1.40	1.77	2.30	3.03	3.64
	global 2	1.15	1.17	1.20	1.25	1.35	1.57	2.05	2.60	3.28	3.78
DLX 2	global 1	1.02	1.03	1.05	1.08	1.14	1.23	1.43	1.93	2.77	3.58
	global 2	1.03	1.04	1.06	1.11	1.19	1.34	1.62	2.22	3.06	3.73
DLX 5	global 1	1.00	1.00	1.00	1.00	1.01	1.03	1.09	1.28	1.96	3.13
	global 2	1.00	1.00	1.00	1.00	1.01	1.04	1.13	1.43	2.28	3.38

Table G.36: *Average number of acceptance tests per aperiodic job for Experiment 5c (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.99	3.97	3.96	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	3.98	3.99	3.98	3.95
	global 2	–	–	–	–	–	–	4.00	4.00	4.00	4.00

Table G.37: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5c (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.17	1.18	1.20	1.23	1.30	1.40	1.49	1.57	1.63	1.67
	global 2	1.25	1.28	1.32	1.37	1.51	1.70	1.96	2.14	2.40	2.41
DLX 1.5	global 1	1.11	1.12	1.14	1.17	1.22	1.32	1.47	1.58	1.69	1.72
	global 2	1.14	1.16	1.19	1.24	1.33	1.50	1.79	2.02	2.28	2.43
DLX 2	global 1	1.02	1.03	1.05	1.08	1.13	1.21	1.34	1.52	1.67	1.73
	global 2	1.03	1.04	1.06	1.11	1.19	1.32	1.52	1.87	2.21	2.42
DLX 5	global 1	1.00	1.00	1.00	1.00	1.01	1.03	1.09	1.23	1.51	1.72
	global 2	1.00	1.00	1.00	1.00	1.01	1.04	1.13	1.38	1.89	2.32

Table G.38: *Average number of acceptance tests per accepted aperiodic job for Experiment 5c (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.71	1.74	1.78	1.85	1.96	2.10	2.31	2.54	2.74	2.92
	global 2	2.02	2.12	2.22	2.33	2.55	2.78	3.09	3.41	3.68	3.90
DLX 1.5	global 1	1.64	1.70	1.76	1.85	2.00	2.24	2.58	2.99	3.39	3.74
	global 2	1.83	1.90	2.02	2.19	2.35	2.62	2.94	3.27	3.61	3.87
DLX 2	global 1	1.54	1.60	1.66	1.76	1.91	2.13	2.44	2.83	3.32	3.70
	global 2	1.69	1.78	1.89	2.19	2.24	2.51	2.81	3.16	3.54	3.85
DLX 5	global 1	1.53	1.58	1.63	1.68	1.75	1.96	2.34	2.75	3.13	3.56
	global 2	1.62	1.72	1.76	1.88	1.99	2.31	2.71	3.09	3.40	3.74

Table G.39: *Average number of acceptance tests per aperiodic job for Experiment 5a (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.94	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.96	3.95	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	4.00	4.00	4.00	3.99	3.98	3.97	3.93
	global 2	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Table G.40: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5a (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.60	1.61	1.63	1.65	1.67	1.68	1.69	1.70	1.71	1.71
	global 2	1.89	1.97	2.04	2.08	2.19	2.25	2.32	2.40	2.45	2.47
DLX 1.5	global 1	1.60	1.64	1.67	1.70	1.73	1.76	1.78	1.78	1.79	1.76
	global 2	1.79	1.84	1.93	2.04	2.11	2.23	2.33	2.38	2.46	2.49
DLX 2	global 1	1.54	1.59	1.64	1.70	1.74	1.77	1.79	1.81	1.80	1.78
	global 2	1.69	1.77	1.86	2.13	2.07	2.20	2.30	2.40	2.44	2.50
DLX 5	global 1	1.53	1.58	1.63	1.68	1.73	1.78	1.81	1.82	1.81	1.82
	global 2	1.62	1.72	1.76	1.87	1.96	2.14	2.29	2.39	2.46	2.51

Table G.41: *Average number of acceptance tests per accepted aperiodic job for Experiment 5a (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.64	1.66	1.71	1.78	1.89	2.05	2.27	2.52	2.74	2.92
	global 2	1.85	1.88	1.99	2.12	2.34	2.63	2.99	3.36	3.66	3.90
DLX 1.5	global 1	1.48	1.52	1.59	1.71	1.89	2.16	2.53	2.95	3.39	3.73
	global 2	1.57	1.63	1.75	1.91	2.13	2.43	2.82	3.19	3.57	3.86
DLX 2	global 1	1.29	1.34	1.42	1.55	1.74	2.01	2.37	2.80	3.30	3.70
	global 2	1.35	1.43	1.55	1.72	1.96	2.27	2.64	3.05	3.49	3.83
DLX 5	global 1	1.11	1.18	1.26	1.35	1.49	1.78	2.22	2.69	3.10	3.56
	global 2	1.14	1.24	1.35	1.49	1.70	2.05	2.51	2.95	3.33	3.72

Table G.42: *Average number of acceptance tests per aperiodic job for Experiment 5b (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.94	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.95	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	4.00	3.99	4.00	4.00	3.99	3.99	3.98	3.97	3.93
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Table G.43: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5b (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.51	1.52	1.54	1.56	1.58	1.60	1.62	1.64	1.67	1.70
	global 2	1.69	1.70	1.76	1.82	1.92	2.03	2.13	2.22	2.31	2.37
DLX 1.5	global 1	1.42	1.44	1.47	1.52	1.58	1.63	1.69	1.72	1.74	1.74
	global 2	1.51	1.56	1.63	1.73	1.84	1.97	2.13	2.21	2.32	2.39
DLX 2	global 1	1.28	1.32	1.38	1.46	1.54	1.61	1.68	1.73	1.75	1.76
	global 2	1.34	1.41	1.51	1.63	1.76	1.90	2.04	2.18	2.30	2.38
DLX 5	global 1	1.11	1.18	1.26	1.35	1.44	1.55	1.65	1.71	1.75	1.79
	global 2	1.14	1.24	1.35	1.49	1.65	1.82	2.01	2.15	2.28	2.38

Table G.44: *Average number of acceptance tests per accepted aperiodic job for Experiment 5b (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.48	1.51	1.57	1.66	1.80	2.00	2.25	2.50	2.73	2.92
	global 2	1.75	1.79	1.91	2.06	2.28	2.61	2.97	3.34	3.66	3.90
DLX 1.5	global 1	1.34	1.39	1.47	1.60	1.80	2.10	2.49	2.94	3.38	3.74
	global 2	1.44	1.51	1.67	1.78	2.05	2.37	2.76	3.19	3.57	3.87
DLX 2	global 1	1.14	1.18	1.26	1.40	1.62	1.93	2.32	2.78	3.30	3.71
	global 2	1.18	1.25	1.35	1.54	1.80	2.18	2.58	3.04	3.49	3.84
DLX 5	global 1	1.01	1.03	1.06	1.12	1.25	1.59	2.13	2.64	3.09	3.55
	global 2	1.02	1.04	1.08	1.17	1.38	1.79	2.38	2.91	3.32	3.72

Table G.45: *Average number of acceptance tests per aperiodic job for Experiment 5c (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.96	3.95	3.94	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.94	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	4.00	4.00	4.00	3.99	3.99	3.98	3.96	3.93
	global 2	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Table G.46: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5c (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.36	1.37	1.39	1.43	1.47	1.53	1.57	1.61	1.64	1.68
	global 2	1.57	1.60	1.67	1.74	1.83	1.99	2.08	2.20	2.30	2.38
DLX 1.5	global 1	1.27	1.30	1.34	1.40	1.47	1.55	1.63	1.68	1.72	1.74
	global 2	1.38	1.43	1.55	1.59	1.74	1.88	2.04	2.19	2.32	2.41
DLX 2	global 1	1.13	1.17	1.22	1.30	1.40	1.51	1.60	1.68	1.73	1.75
	global 2	1.17	1.23	1.31	1.44	1.59	1.79	1.96	2.14	2.28	2.40
DLX 5	global 1	1.01	1.03	1.06	1.12	1.21	1.35	1.51	1.63	1.70	1.77
	global 2	1.02	1.04	1.08	1.16	1.33	1.56	1.84	2.07	2.24	2.40

Table G.47: *Average number of acceptance tests per accepted aperiodic job for Experiment 5c (50% aperiodic job utilization).*

G.4 Average Number of Acceptance Tests with SDL

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.25	1.26	1.28	1.32	1.39	1.52	1.78	2.10	2.48	2.82
	global 2	1.27	1.28	1.31	1.36	1.47	1.73	2.20	2.73	3.30	3.77
DLX 1.5	global 1	1.17	1.18	1.21	1.24	1.30	1.41	1.63	2.08	2.78	3.57
	global 2	1.17	1.19	1.21	1.25	1.32	1.48	1.80	2.35	3.04	3.70
DLX 2	global 1	1.06	1.08	1.11	1.16	1.22	1.30	1.42	1.70	2.43	3.45
	global 2	1.06	1.08	1.11	1.16	1.23	1.32	1.49	1.90	2.71	3.61
DLX 5	global 1	1.00	1.00	1.00	1.02	1.04	1.11	1.22	1.38	1.62	2.72
	global 2	1.00	1.00	1.00	1.02	1.04	1.11	1.22	1.40	1.76	3.01

Table G.48: *Average number of acceptance tests per aperiodic job for Experiment 5a (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.99	3.99	3.98	3.97
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	–	–	–	4.00	4.00	4.00	3.99	3.98	3.97
	global 2	–	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	–	4.00	3.99	3.97
	global 2	–	–	–	–	–	–	–	4.00	4.00	4.00

Table G.49: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5a (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.25	1.26	1.28	1.32	1.37	1.44	1.52	1.59	1.66	1.70
	global 2	1.27	1.28	1.30	1.36	1.46	1.64	1.87	2.08	2.27	2.38
DLX 1.5	global 1	1.17	1.18	1.21	1.24	1.29	1.39	1.50	1.62	1.70	1.72
	global 2	1.17	1.19	1.21	1.25	1.32	1.46	1.70	1.98	2.21	2.35
DLX 2	global 1	1.06	1.08	1.11	1.16	1.22	1.30	1.41	1.56	1.69	1.73
	global 2	1.06	1.08	1.11	1.16	1.23	1.32	1.47	1.78	2.13	2.32
DLX 5	global 1	1.00	1.00	1.00	1.02	1.04	1.11	1.22	1.38	1.58	1.76
	global 2	1.00	1.00	1.00	1.02	1.04	1.11	1.22	1.40	1.72	2.26

Table G.50: *Average number of acceptance tests per accepted aperiodic job for Experiment 5a (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.16	1.17	1.19	1.22	1.30	1.46	1.74	2.06	2.47	2.83
	global 2	1.16	1.17	1.19	1.24	1.36	1.64	2.13	2.66	3.28	3.78
DLX 1.5	global 1	1.09	1.10	1.11	1.14	1.18	1.29	1.54	2.03	2.76	3.57
	global 2	1.09	1.10	1.12	1.15	1.20	1.35	1.68	2.27	3.00	3.70
DLX 2	global 1	1.02	1.03	1.04	1.07	1.10	1.17	1.29	1.61	2.42	3.46
	global 2	1.02	1.03	1.04	1.07	1.11	1.18	1.33	1.76	2.68	3.61
DLX 5	global 1	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.16	1.44	2.69
	global 2	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.18	1.56	2.96

Table G.51: *Average number of acceptance tests per aperiodic job for Experiment 5b (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	–	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.98	3.97
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	–	–	–	4.00	3.97	4.00	3.98	3.98	3.97
	global 2	–	–	–	–	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	–	4.00	3.99	3.96
	global 2	–	–	–	–	–	–	–	4.00	4.00	4.00

Table G.52: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5b (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.16	1.17	1.18	1.22	1.28	1.38	1.49	1.56	1.63	1.69
	global 2	1.16	1.17	1.19	1.23	1.34	1.54	1.80	2.01	2.22	2.39
DLX 1.5	global 1	1.09	1.10	1.11	1.14	1.18	1.27	1.41	1.56	1.67	1.73
	global 2	1.09	1.10	1.12	1.15	1.19	1.33	1.57	1.89	2.15	2.37
DLX 2	global 1	1.02	1.03	1.04	1.07	1.10	1.17	1.27	1.46	1.65	1.73
	global 2	1.02	1.03	1.04	1.07	1.11	1.18	1.32	1.64	2.07	2.32
DLX 5	global 1	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.16	1.39	1.70
	global 2	1.00	1.00	1.00	1.00	1.00	1.01	1.05	1.18	1.51	2.17

Table G.53: *Average number of acceptance tests per accepted aperiodic job for Experiment 5b (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.10	1.10	1.11	1.14	1.22	1.40	1.70	2.06	2.46	2.82
	global 2	1.11	1.11	1.13	1.18	1.29	1.57	2.08	2.66	3.27	3.77
DLX 1.5	global 1	1.05	1.06	1.07	1.08	1.12	1.22	1.48	2.00	2.76	3.56
	global 2	1.06	1.06	1.07	1.09	1.13	1.28	1.62	2.23	3.00	3.69
DLX 2	global 1	1.00	1.01	1.02	1.03	1.06	1.10	1.20	1.53	2.37	3.45
	global 2	1.00	1.01	1.02	1.04	1.06	1.11	1.25	1.68	2.62	3.60
DLX 5	global 1	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.07	1.32	2.67
	global 2	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.08	1.43	2.94

Table G.54: *Average number of acceptance tests per aperiodic job for Experiment 5c (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.98	3.97
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	–	–	4.00	4.00	4.00	3.99	3.98	3.97	3.97
	global 2	–	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	–	4.00	3.99	3.96
	global 2	–	–	–	–	–	–	–	4.00	4.00	4.00

Table G.55: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5c (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.09	1.10	1.11	1.14	1.20	1.31	1.44	1.54	1.61	1.68
	global 2	1.11	1.11	1.13	1.17	1.27	1.47	1.75	1.99	2.17	2.33
DLX 1.5	global 1	1.05	1.06	1.07	1.08	1.11	1.20	1.36	1.52	1.65	1.72
	global 2	1.06	1.06	1.07	1.09	1.13	1.26	1.52	1.83	2.12	2.33
DLX 2	global 1	1.00	1.01	1.02	1.03	1.06	1.10	1.19	1.38	1.61	1.72
	global 2	1.00	1.01	1.02	1.04	1.06	1.11	1.23	1.55	2.00	2.31
DLX 5	global 1	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.07	1.27	1.67
	global 2	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.08	1.37	2.13

Table G.56: *Average number of acceptance tests per accepted aperiodic job for Experiment 5c (10% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.41	1.42	1.45	1.50	1.57	1.72	1.97	2.26	2.59	2.87
	global 2	1.47	1.49	1.53	1.61	1.76	2.06	2.50	2.97	3.46	3.83
DLX 1.5	global 1	1.31	1.33	1.37	1.42	1.49	1.63	1.91	2.42	3.04	3.65
	global 2	1.33	1.36	1.41	1.47	1.58	1.79	2.16	2.71	3.28	3.78
DLX 2	global 1	1.19	1.23	1.28	1.34	1.42	1.52	1.68	2.07	2.82	3.58
	global 2	1.19	1.24	1.29	1.36	1.47	1.61	1.86	2.34	3.09	3.72
DLX 5	global 1	1.04	1.07	1.13	1.22	1.31	1.42	1.53	1.68	2.13	3.16
	global 2	1.04	1.07	1.13	1.22	1.31	1.43	1.58	1.84	2.45	3.39

Table G.57: *Average number of acceptance tests per aperiodic job for Experiment 5a (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.99	3.97	3.97	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.97	3.96
	global 2	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	4.00	4.00	3.98	3.95
	global 2	–	–	–	–	–	4.00	4.00	4.00	4.00	4.00

Table G.58: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5a (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.40	1.41	1.44	1.47	1.52	1.57	1.61	1.65	1.68	1.70
	global 2	1.47	1.48	1.52	1.58	1.69	1.87	2.04	2.19	2.33	2.39
DLX 1.5	global 1	1.31	1.33	1.37	1.41	1.47	1.55	1.64	1.70	1.74	1.75
	global 2	1.33	1.36	1.41	1.47	1.56	1.72	1.94	2.14	2.29	2.43
DLX 2	global 1	1.19	1.23	1.28	1.34	1.42	1.50	1.60	1.69	1.77	1.77
	global 2	1.19	1.24	1.29	1.36	1.47	1.59	1.78	2.03	2.28	2.41
DLX 5	global 1	1.04	1.07	1.13	1.22	1.31	1.42	1.53	1.65	1.73	1.80
	global 2	1.04	1.07	1.13	1.22	1.31	1.43	1.58	1.79	2.10	2.35

Table G.59: *Average number of acceptance tests per accepted aperiodic job for Experiment 5a (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.31	1.32	1.35	1.39	1.48	1.65	1.93	2.25	2.58	2.86
	global 2	1.33	1.35	1.38	1.46	1.61	1.93	2.43	2.94	3.44	3.82
DLX 1.5	global 1	1.19	1.20	1.23	1.27	1.34	1.50	1.82	2.36	3.02	3.64
	global 2	1.20	1.21	1.25	1.31	1.40	1.62	2.03	2.62	3.25	3.77
DLX 2	global 1	1.06	1.08	1.11	1.16	1.23	1.34	1.54	1.99	2.80	3.58
	global 2	1.06	1.09	1.12	1.18	1.27	1.42	1.68	2.23	3.05	3.72
DLX 5	global 1	1.00	1.00	1.01	1.02	1.04	1.11	1.23	1.45	2.03	3.14
	global 2	1.00	1.00	1.01	1.02	1.05	1.13	1.28	1.60	2.31	3.37

Table G.60: *Average number of acceptance tests per aperiodic job for Experiment 5b (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.96	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.97	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	4.00	3.99	3.99	3.98	3.95
	global 2	–	–	–	–	–	4.00	4.00	4.00	4.00	4.00

Table G.61: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5b (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.29	1.30	1.32	1.36	1.41	1.48	1.54	1.60	1.65	1.69
	global 2	1.32	1.33	1.36	1.42	1.54	1.72	1.93	2.10	2.25	2.36
DLX 1.5	global 1	1.19	1.20	1.22	1.26	1.32	1.41	1.53	1.63	1.70	1.73
	global 2	1.19	1.21	1.25	1.30	1.38	1.54	1.78	2.02	2.22	2.36
DLX 2	global 1	1.06	1.08	1.11	1.16	1.23	1.32	1.44	1.59	1.71	1.75
	global 2	1.06	1.09	1.12	1.18	1.26	1.39	1.59	1.88	2.18	2.35
DLX 5	global 1	1.00	1.00	1.01	1.02	1.04	1.11	1.23	1.40	1.61	1.76
	global 2	1.00	1.00	1.01	1.02	1.05	1.13	1.28	1.55	1.92	2.29

Table G.62: *Average number of acceptance tests per accepted aperiodic job for Experiment 5b (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.19	1.20	1.22	1.26	1.37	1.57	1.89	2.23	2.58	2.86
	global 2	1.24	1.25	1.29	1.35	1.53	1.86	2.38	2.92	3.44	3.82
DLX 1.5	global 1	1.11	1.12	1.14	1.18	1.24	1.40	1.77	2.30	3.03	3.64
	global 2	1.12	1.14	1.17	1.21	1.30	1.52	1.97	2.55	3.25	3.77
DLX 2	global 1	1.02	1.04	1.05	1.09	1.14	1.23	1.43	1.93	2.77	3.57
	global 2	1.02	1.04	1.06	1.10	1.16	1.29	1.56	2.16	3.02	3.71
DLX 5	global 1	1.00	1.00	1.00	1.00	1.01	1.03	1.09	1.29	1.96	3.13
	global 2	1.00	1.00	1.00	1.00	1.01	1.03	1.11	1.38	2.21	3.35

Table G.63: *Average number of acceptance tests per aperiodic job for Experiment 5c (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.96	3.95
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	–	4.00	4.00	3.98	4.00	4.00	3.99	3.97	3.97	3.95
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	–	–	–	–	3.98	3.99	3.98	3.95
	global 2	–	–	–	–	–	–	4.00	4.00	4.00	4.00

Table G.64: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5c (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.18	1.18	1.20	1.23	1.30	1.40	1.50	1.57	1.63	1.67
	global 2	1.23	1.23	1.27	1.32	1.44	1.64	1.88	2.06	2.23	2.33
DLX 1.5	global 1	1.11	1.12	1.14	1.17	1.22	1.32	1.46	1.58	1.68	1.72
	global 2	1.12	1.14	1.16	1.20	1.28	1.44	1.71	1.95	2.19	2.35
DLX 2	global 1	1.02	1.04	1.05	1.09	1.13	1.21	1.33	1.51	1.67	1.73
	global 2	1.02	1.04	1.06	1.10	1.16	1.27	1.46	1.79	2.13	2.34
DLX 5	global 1	1.00	1.00	1.00	1.00	1.01	1.03	1.09	1.24	1.50	1.72
	global 2	1.00	1.00	1.00	1.00	1.01	1.03	1.11	1.33	1.79	2.24

Table G.65: *Average number of acceptance tests per accepted aperiodic job for Experiment 5c (20% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.70	1.73	1.77	1.84	1.95	2.10	2.31	2.54	2.74	2.92
	global 2	1.96	2.00	2.09	2.23	2.43	2.71	3.04	3.38	3.67	3.90
DLX 1.5	global 1	1.63	1.66	1.74	1.83	1.98	2.23	2.57	2.98	3.39	3.74
	global 2	1.74	1.80	1.91	2.05	2.25	2.53	2.86	3.23	3.58	3.87
DLX 2	global 1	1.54	1.59	1.66	1.74	1.88	2.11	2.43	2.83	3.31	3.70
	global 2	1.59	1.67	1.76	1.89	2.12	2.39	2.73	3.10	3.51	3.84
DLX 5	global 1	1.54	1.59	1.63	1.68	1.75	1.95	2.33	2.74	3.13	3.56
	global 2	1.54	1.59	1.65	1.72	1.86	2.18	2.61	3.02	3.36	3.72

Table G.66: *Average number of acceptance tests per aperiodic job for Experiment 5a (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.94	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.96	3.95	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.93
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Table G.67: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5a (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.60	1.62	1.63	1.65	1.67	1.68	1.69	1.70	1.71	1.71
	global 2	1.83	1.84	1.89	1.96	2.05	2.15	2.23	2.32	2.39	2.41
DLX 1.5	global 1	1.58	1.61	1.64	1.68	1.71	1.74	1.77	1.78	1.78	1.76
	global 2	1.70	1.74	1.81	1.89	1.99	2.11	2.21	2.30	2.37	2.43
DLX 2	global 1	1.54	1.59	1.63	1.68	1.72	1.76	1.78	1.80	1.80	1.78
	global 2	1.59	1.65	1.73	1.82	1.95	2.07	2.18	2.28	2.37	2.44
DLX 5	global 1	1.54	1.59	1.63	1.68	1.73	1.77	1.80	1.80	1.80	1.82
	global 2	1.54	1.59	1.65	1.72	1.83	1.99	2.15	2.27	2.35	2.42

Table G.68: *Average number of acceptance tests per accepted aperiodic job for Experiment 5a (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.64	1.66	1.71	1.78	1.89	2.05	2.28	2.52	2.74	2.92
	global 2	1.84	1.87	1.96	2.11	2.32	2.63	2.98	3.35	3.66	3.90
DLX 1.5	global 1	1.48	1.51	1.59	1.70	1.89	2.16	2.53	2.95	3.39	3.73
	global 2	1.56	1.62	1.73	1.90	2.12	2.42	2.78	3.18	3.57	3.86
DLX 2	global 1	1.29	1.34	1.43	1.55	1.74	2.01	2.37	2.80	3.30	3.70
	global 2	1.34	1.43	1.54	1.71	1.94	2.26	2.63	3.04	3.49	3.83
DLX 5	global 1	1.11	1.18	1.26	1.36	1.49	1.78	2.22	2.69	3.10	3.56
	global 2	1.14	1.24	1.35	1.48	1.68	2.01	2.49	2.94	3.32	3.71

Table G.69: *Average number of acceptance tests per aperiodic job for Experiment 5b (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.94	3.91
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.94	3.91
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	4.00	4.00	4.00	4.00	3.99	3.99	3.98	3.96	3.93
	global 2	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Table G.70: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5b (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.51	1.52	1.54	1.56	1.58	1.60	1.62	1.64	1.67	1.70
	global 2	1.68	1.69	1.74	1.81	1.90	2.02	2.11	2.21	2.33	2.39
DLX 1.5	global 1	1.42	1.44	1.47	1.52	1.57	1.63	1.68	1.72	1.74	1.74
	global 2	1.50	1.55	1.62	1.71	1.82	1.96	2.08	2.19	2.31	2.39
DLX 2	global 1	1.28	1.33	1.39	1.46	1.53	1.61	1.67	1.73	1.75	1.76
	global 2	1.33	1.41	1.49	1.61	1.74	1.89	2.04	2.16	2.29	2.38
DLX 5	global 1	1.11	1.18	1.26	1.35	1.45	1.55	1.64	1.71	1.74	1.79
	global 2	1.14	1.24	1.35	1.47	1.62	1.79	1.98	2.14	2.25	2.37

Table G.71: *Average number of acceptance tests per accepted aperiodic job for Experiment 5b (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.48	1.51	1.57	1.66	1.80	2.00	2.25	2.50	2.73	2.92
	global 2	1.74	1.78	1.87	2.02	2.26	2.58	2.96	3.33	3.65	3.89
DLX 1.5	global 1	1.33	1.38	1.46	1.59	1.80	2.10	2.49	2.94	3.37	3.73
	global 2	1.44	1.49	1.60	1.76	2.00	2.35	2.75	3.17	3.56	3.86
DLX 2	global 1	1.14	1.19	1.27	1.40	1.62	1.93	2.32	2.78	3.30	3.70
	global 2	1.17	1.25	1.35	1.53	1.80	2.15	2.57	3.02	3.49	3.84
DLX 5	global 1	1.02	1.04	1.07	1.12	1.25	1.59	2.13	2.64	3.09	3.55
	global 2	1.02	1.04	1.08	1.16	1.36	1.77	2.36	2.88	3.31	3.71

Table G.72: *Average number of acceptance tests per aperiodic job for Experiment 5c (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 1.5	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.96	3.95	3.94	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 2	global 1	4.00	4.00	4.00	4.00	3.99	3.98	3.97	3.95	3.94	3.92
	global 2	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00
DLX 5	global 1	–	–	4.00	4.00	4.00	3.99	3.99	3.98	3.96	3.93
	global 2	–	–	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

Table G.73: *Average number of acceptance tests per finally rejected aperiodic job for Experiment 5c (50% aperiodic job utilization).*

Name		Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	1.36	1.37	1.39	1.43	1.47	1.53	1.57	1.61	1.64	1.68
	global 2	1.56	1.58	1.63	1.70	1.81	1.94	2.06	2.17	2.27	2.34
DLX 1.5	global 1	1.27	1.30	1.34	1.40	1.46	1.55	1.62	1.68	1.72	1.74
	global 2	1.37	1.41	1.48	1.57	1.69	1.85	2.01	2.14	2.28	2.38
DLX 2	global 1	1.13	1.18	1.23	1.30	1.40	1.50	1.60	1.67	1.73	1.75
	global 2	1.17	1.23	1.31	1.42	1.58	1.76	1.95	2.10	2.25	2.38
DLX 5	global 1	1.02	1.04	1.07	1.12	1.21	1.35	1.51	1.63	1.70	1.77
	global 2	1.02	1.04	1.08	1.16	1.31	1.54	1.80	2.03	2.20	2.35

Table G.74: *Average number of acceptance tests per accepted aperiodic job for Experiment 5c (50% aperiodic job utilization).*

G.5 Quickness

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.73	0.72	0.70	0.66	0.60	0.52	0.44	0.37	0.31	0.26
	global 2	0.72	0.72	0.69	0.66	0.60	0.52	0.43	0.35	0.30	0.26
DLX 1.5	global 1	0.88	0.87	0.85	0.83	0.80	0.75	0.69	0.64	0.62	0.62
	global 2	0.88	0.87	0.85	0.83	0.79	0.74	0.68	0.64	0.62	0.62
DLX 2	global 1	0.83	0.80	0.76	0.73	0.69	0.63	0.57	0.50	0.47	0.51
	global 2	0.83	0.80	0.76	0.72	0.67	0.62	0.55	0.48	0.46	0.50
DLX 5	global 1	0.92	0.87	0.81	0.72	0.62	0.51	0.41	0.32	0.22	0.15
	global 2	0.92	0.87	0.81	0.72	0.61	0.50	0.40	0.30	0.20	0.15

Table G.75: *Linux Experiment 5a: average quickness,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.57	0.55	0.53	0.50	0.45	0.39	0.34	0.31	0.28	0.26
	global 2	0.56	0.55	0.53	0.49	0.44	0.38	0.33	0.30	0.27	0.25
DLX 1.5	global 1	0.81	0.79	0.78	0.77	0.74	0.70	0.64	0.60	0.58	0.59
	global 2	0.80	0.79	0.77	0.76	0.73	0.69	0.64	0.60	0.59	0.59
DLX 2	global 1	0.72	0.70	0.68	0.66	0.63	0.59	0.52	0.46	0.44	0.48
	global 2	0.71	0.69	0.66	0.63	0.61	0.56	0.50	0.44	0.43	0.48
DLX 5	global 1	0.71	0.62	0.54	0.49	0.47	0.45	0.40	0.30	0.18	0.14
	global 2	0.71	0.62	0.54	0.48	0.45	0.41	0.35	0.26	0.17	0.13

Table G.76: *Linux Experiment 5a: average quickness,* $U_{aperiodic} = 20\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.34	0.33	0.31	0.29	0.28	0.26	0.25	0.24	0.24	0.23
	global 2	0.33	0.32	0.31	0.29	0.27	0.26	0.25	0.24	0.23	0.23
DLX 1.5	global 1	0.69	0.68	0.67	0.65	0.61	0.57	0.53	0.51	0.51	0.53
	global 2	0.66	0.65	0.63	0.62	0.60	0.57	0.54	0.52	0.52	0.53
DLX 2	global 1	0.63	0.61	0.59	0.56	0.50	0.45	0.39	0.36	0.37	0.43
	global 2	0.57	0.55	0.53	0.50	0.47	0.43	0.38	0.36	0.37	0.43
DLX 5	global 1	0.58	0.59	0.59	0.56	0.47	0.30	0.17	0.12	0.11	0.12
	global 2	0.55	0.54	0.52	0.47	0.37	0.25	0.16	0.12	0.11	0.12

Table G.77: *Linux Experiment 5a: average quickness,* $U_{aperiodic} = 50\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.82	0.82	0.80	0.76	0.70	0.60	0.47	0.40	0.33	0.26
	global 2	0.84	0.83	0.81	0.78	0.71	0.60	0.47	0.39	0.32	0.26
DLX 1.5	global 1	0.93	0.92	0.90	0.88	0.84	0.78	0.71	0.65	0.62	0.62
	global 2	0.93	0.92	0.90	0.88	0.83	0.77	0.70	0.65	0.62	0.62
DLX 2	global 1	0.91	0.88	0.84	0.79	0.73	0.66	0.58	0.50	0.47	0.51
	global 2	0.91	0.88	0.84	0.78	0.73	0.66	0.57	0.49	0.47	0.51
DLX 5	global 1	0.97	0.93	0.89	0.83	0.75	0.64	0.49	0.35	0.22	0.15
	global 2	0.97	0.93	0.89	0.83	0.75	0.64	0.49	0.34	0.21	0.15

Table G.78: *Linux Experiment 5b: average quickness, $U_{aperiodic} = 10\%$.*

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.68	0.67	0.65	0.61	0.56	0.49	0.41	0.35	0.31	0.26
	global 2	0.71	0.70	0.68	0.64	0.58	0.49	0.41	0.34	0.30	0.26
DLX 1.5	global 1	0.87	0.86	0.84	0.82	0.79	0.73	0.66	0.61	0.58	0.59
	global 2	0.87	0.86	0.84	0.81	0.77	0.72	0.66	0.62	0.59	0.60
DLX 2	global 1	0.83	0.79	0.76	0.72	0.67	0.61	0.53	0.46	0.44	0.48
	global 2	0.83	0.79	0.75	0.71	0.66	0.60	0.52	0.45	0.43	0.48
DLX 5	global 1	0.92	0.87	0.80	0.72	0.61	0.50	0.39	0.28	0.18	0.14
	global 2	0.92	0.87	0.80	0.72	0.61	0.48	0.37	0.26	0.17	0.13

Table G.79: *Linux Experiment 5b: average quickness, $U_{aperiodic} = 20\%$.*

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.43	0.42	0.41	0.39	0.37	0.35	0.32	0.30	0.27	0.24
	global 2	0.47	0.46	0.44	0.42	0.38	0.35	0.31	0.29	0.26	0.24
DLX 1.5	global 1	0.73	0.72	0.70	0.68	0.64	0.59	0.54	0.52	0.52	0.54
	global 2	0.72	0.70	0.68	0.65	0.62	0.59	0.55	0.53	0.52	0.54
DLX 2	global 1	0.65	0.63	0.60	0.56	0.51	0.45	0.40	0.37	0.38	0.43
	global 2	0.63	0.59	0.56	0.52	0.48	0.44	0.39	0.36	0.37	0.44
DLX 5	global 1	0.58	0.52	0.47	0.44	0.37	0.26	0.17	0.12	0.11	0.12
	global 2	0.55	0.47	0.42	0.38	0.32	0.24	0.16	0.12	0.11	0.12

Table G.80: *Linux Experiment 5b: average quickness, $U_{aperiodic} = 50\%$.*

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.90	0.89	0.88	0.85	0.78	0.66	0.52	0.43	0.35	0.28
	global 2	0.89	0.89	0.88	0.84	0.77	0.65	0.51	0.41	0.33	0.27
DLX 1.5	global 1	0.96	0.95	0.93	0.91	0.86	0.80	0.72	0.66	0.62	0.62
	global 2	0.96	0.95	0.93	0.90	0.86	0.79	0.71	0.66	0.62	0.62
DLX 2	global 1	0.95	0.92	0.88	0.83	0.76	0.69	0.59	0.51	0.47	0.51
	global 2	0.95	0.92	0.88	0.82	0.76	0.68	0.58	0.50	0.46	0.51
DLX 5	global 1	0.98	0.95	0.91	0.86	0.80	0.70	0.55	0.38	0.22	0.15
	global 2	0.98	0.95	0.91	0.86	0.80	0.70	0.55	0.38	0.22	0.15

Table G.81: *Linux Experiment 5c: average quickness,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.81	0.80	0.78	0.75	0.67	0.57	0.46	0.39	0.33	0.28
	global 2	0.80	0.79	0.76	0.73	0.65	0.55	0.44	0.37	0.31	0.27
DLX 1.5	global 1	0.92	0.91	0.89	0.86	0.82	0.75	0.67	0.62	0.59	0.59
	global 2	0.91	0.90	0.88	0.85	0.81	0.75	0.67	0.62	0.59	0.60
DLX 2	global 1	0.90	0.86	0.82	0.77	0.71	0.64	0.55	0.46	0.44	0.48
	global 2	0.89	0.86	0.82	0.76	0.70	0.63	0.54	0.46	0.44	0.48
DLX 5	global 1	0.96	0.92	0.87	0.81	0.72	0.61	0.46	0.31	0.18	0.13
	global 2	0.96	0.92	0.87	0.81	0.72	0.60	0.45	0.29	0.17	0.13

Table G.82: *Linux Experiment 5c: average quickness,* $U_{aperiodic} = 20\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.60	0.59	0.57	0.53	0.48	0.42	0.37	0.33	0.30	0.27
	global 2	0.57	0.56	0.54	0.50	0.45	0.40	0.35	0.30	0.28	0.26
DLX 1.5	global 1	0.80	0.78	0.75	0.71	0.66	0.60	0.55	0.53	0.52	0.54
	global 2	0.77	0.75	0.72	0.69	0.65	0.60	0.56	0.53	0.53	0.54
DLX 2	global 1	0.75	0.70	0.65	0.59	0.53	0.46	0.40	0.36	0.38	0.44
	global 2	0.73	0.68	0.63	0.57	0.51	0.45	0.40	0.36	0.38	0.43
DLX 5	global 1	0.82	0.74	0.66	0.56	0.43	0.28	0.18	0.13	0.11	0.12
	global 2	0.81	0.73	0.64	0.52	0.40	0.26	0.17	0.12	0.11	0.12

Table G.83: *Linux Experiment 5c: average quickness,* $U_{aperiodic} = 50\%$.

G.6 Quickness with SDL

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.73	0.72	0.70	0.66	0.60	0.52	0.44	0.37	0.32	0.26
	global 2	0.72	0.72	0.69	0.66	0.60	0.52	0.43	0.35	0.30	0.26
DLX 1.5	global 1	0.88	0.88	0.88	0.87	0.84	0.79	0.73	0.68	0.65	0.65
	global 2	0.88	0.88	0.88	0.87	0.83	0.78	0.72	0.67	0.65	0.65
DLX 2	global 1	0.79	0.82	0.83	0.83	0.82	0.78	0.72	0.64	0.58	0.62
	global 2	0.79	0.82	0.82	0.82	0.81	0.77	0.69	0.61	0.57	0.62
DLX 5	global 1	0.88	0.89	0.87	0.83	0.77	0.70	0.66	0.64	0.57	0.35
	global 2	0.88	0.89	0.87	0.83	0.77	0.70	0.64	0.60	0.52	0.34

Table G.84: *Linux Experiment 5a with SDL: average quickness,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.56	0.55	0.53	0.49	0.45	0.39	0.34	0.31	0.28	0.26
	global 2	0.56	0.55	0.53	0.49	0.44	0.38	0.33	0.30	0.27	0.25
DLX 1.5	global 1	0.81	0.81	0.81	0.80	0.78	0.74	0.68	0.63	0.61	0.62
	global 2	0.80	0.80	0.80	0.79	0.77	0.73	0.67	0.63	0.61	0.62
DLX 2	global 1	0.69	0.72	0.73	0.74	0.75	0.73	0.66	0.58	0.53	0.57
	global 2	0.67	0.71	0.72	0.72	0.72	0.69	0.62	0.55	0.53	0.57
DLX 5	global 1	0.65	0.65	0.61	0.59	0.59	0.61	0.63	0.59	0.42	0.28
	global 2	0.65	0.65	0.61	0.58	0.57	0.56	0.54	0.49	0.38	0.28

Table G.85: *Linux Experiment 5a with SDL: average quickness,* $U_{aperiodic} = 20\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.33	0.32	0.31	0.29	0.28	0.26	0.25	0.24	0.24	0.23
	global 2	0.33	0.32	0.31	0.29	0.27	0.26	0.25	0.24	0.23	0.23
DLX 1.5	global 1	0.69	0.69	0.69	0.68	0.65	0.60	0.56	0.53	0.53	0.55
	global 2	0.66	0.66	0.66	0.65	0.63	0.60	0.56	0.54	0.54	0.56
DLX 2	global 1	0.62	0.65	0.65	0.64	0.61	0.55	0.48	0.43	0.43	0.49
	global 2	0.55	0.58	0.58	0.58	0.55	0.51	0.46	0.42	0.43	0.50
DLX 5	global 1	0.58	0.62	0.65	0.66	0.60	0.44	0.28	0.20	0.17	0.19
	global 2	0.52	0.56	0.55	0.54	0.47	0.36	0.25	0.19	0.17	0.19

Table G.86: *Linux Experiment 5a with SDL: average quickness,* $U_{aperiodic} = 50\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.82	0.82	0.80	0.76	0.70	0.60	0.48	0.40	0.33	0.26
	global 2	0.84	0.83	0.81	0.78	0.71	0.60	0.47	0.40	0.32	0.26
DLX 1.5	global 1	0.93	0.93	0.93	0.92	0.88	0.82	0.75	0.69	0.65	0.66
	global 2	0.93	0.93	0.93	0.92	0.88	0.82	0.75	0.69	0.66	0.66
DLX 2	global 1	0.88	0.90	0.90	0.90	0.88	0.82	0.73	0.64	0.59	0.61
	global 2	0.88	0.90	0.90	0.89	0.87	0.81	0.72	0.62	0.58	0.61
DLX 5	global 1	0.95	0.96	0.95	0.94	0.91	0.86	0.77	0.69	0.58	0.35
	global 2	0.95	0.96	0.95	0.94	0.91	0.86	0.77	0.67	0.54	0.34

Table G.87: *Linux Experiment 5b with SDL: average quickness,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.68	0.67	0.65	0.61	0.56	0.49	0.41	0.35	0.31	0.26
	global 2	0.71	0.70	0.68	0.64	0.58	0.49	0.41	0.34	0.30	0.26
DLX 1.5	global 1	0.87	0.87	0.87	0.86	0.83	0.77	0.70	0.65	0.61	0.63
	global 2	0.87	0.87	0.86	0.85	0.82	0.76	0.70	0.65	0.62	0.63
DLX 2	global 1	0.79	0.82	0.82	0.82	0.80	0.76	0.68	0.58	0.53	0.57
	global 2	0.79	0.81	0.81	0.81	0.79	0.74	0.66	0.57	0.53	0.57
DLX 5	global 1	0.88	0.90	0.87	0.83	0.77	0.70	0.64	0.56	0.41	0.28
	global 2	0.88	0.90	0.87	0.83	0.76	0.68	0.60	0.51	0.39	0.28

Table G.88: *Linux Experiment 5b with SDL: average quickness,* $U_{aperiodic} = 20\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.43	0.42	0.41	0.39	0.37	0.35	0.32	0.30	0.27	0.24
	global 2	0.48	0.47	0.45	0.42	0.39	0.35	0.31	0.29	0.26	0.24
DLX 1.5	global 1	0.73	0.73	0.73	0.71	0.67	0.62	0.57	0.55	0.54	0.56
	global 2	0.72	0.72	0.71	0.69	0.66	0.62	0.58	0.55	0.54	0.57
DLX 2	global 1	0.62	0.66	0.65	0.64	0.61	0.56	0.49	0.44	0.44	0.50
	global 2	0.60	0.62	0.61	0.60	0.58	0.54	0.48	0.43	0.44	0.50
DLX 5	global 1	0.54	0.55	0.54	0.53	0.49	0.39	0.27	0.20	0.18	0.20
	global 2	0.51	0.51	0.48	0.47	0.42	0.34	0.25	0.20	0.18	0.19

Table G.89: *Linux Experiment 5b with SDL: average quickness,* $U_{aperiodic} = 50\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.90	0.89	0.88	0.85	0.78	0.66	0.52	0.43	0.35	0.28
	global 2	0.89	0.89	0.88	0.84	0.77	0.65	0.51	0.41	0.33	0.27
DLX 1.5	global 1	0.96	0.96	0.96	0.94	0.91	0.85	0.76	0.70	0.65	0.65
	global 2	0.96	0.96	0.95	0.94	0.90	0.84	0.76	0.69	0.66	0.66
DLX 2	global 1	0.93	0.94	0.94	0.93	0.91	0.85	0.75	0.65	0.59	0.62
	global 2	0.93	0.94	0.94	0.93	0.91	0.85	0.74	0.64	0.58	0.62
DLX 5	global 1	0.97	0.98	0.98	0.97	0.95	0.91	0.84	0.75	0.59	0.35
	global 2	0.97	0.98	0.98	0.97	0.95	0.91	0.84	0.74	0.57	0.35

Table G.90: *Linux Experiment 5c with SDL: average quickness,* $U_{aperiodic} = 10\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.81	0.80	0.78	0.74	0.67	0.57	0.46	0.39	0.33	0.28
	global 2	0.80	0.79	0.77	0.73	0.65	0.55	0.44	0.37	0.31	0.27
DLX 1.5	global 1	0.92	0.92	0.91	0.90	0.86	0.80	0.71	0.66	0.62	0.62
	global 2	0.91	0.91	0.91	0.89	0.85	0.79	0.71	0.66	0.62	0.63
DLX 2	global 1	0.87	0.89	0.88	0.88	0.85	0.79	0.69	0.59	0.54	0.57
	global 2	0.86	0.88	0.88	0.87	0.84	0.78	0.67	0.57	0.53	0.57
DLX 5	global 1	0.93	0.95	0.94	0.92	0.88	0.82	0.73	0.61	0.42	0.28
	global 2	0.93	0.95	0.94	0.92	0.88	0.81	0.72	0.58	0.41	0.28

Table G.91: *Linux Experiment 5c with SDL: average quickness,* $U_{aperiodic} = 20\%$.

Average Quickness	Alg.	Utilization Created by Offline Jobs									
		0%	10%	20%	30%	40%	50%	60%	70%	80%	90%
DLX 1	global 1	0.60	0.59	0.57	0.53	0.48	0.42	0.37	0.33	0.30	0.26
	global 2	0.58	0.56	0.54	0.50	0.45	0.40	0.35	0.31	0.28	0.26
DLX 1.5	global 1	0.80	0.79	0.77	0.75	0.70	0.64	0.59	0.55	0.54	0.56
	global 2	0.77	0.77	0.75	0.73	0.69	0.64	0.59	0.56	0.55	0.56
DLX 2	global 1	0.71	0.73	0.71	0.69	0.64	0.57	0.49	0.44	0.44	0.50
	global 2	0.69	0.70	0.69	0.66	0.61	0.55	0.48	0.43	0.44	0.50
DLX 5	global 1	0.76	0.77	0.72	0.66	0.56	0.41	0.28	0.21	0.18	0.20
	global 2	0.76	0.76	0.70	0.63	0.52	0.38	0.27	0.21	0.18	0.19

Table G.92: *Linux Experiment 5c with SDL: average quickness,* $U_{aperiodic} = 50\%$.

Bibliography

[1] G. Fohler, "Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995, pp. 152–161.

[2] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *Software Engineering, IEEE Transactions on*, vol. 14, no. 2, pp. 141–154, 1988.

[3] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzén, V. Romero, and C. Scordino, "Resource management on multicore systems: The actors approach," *IEEE Micro*, vol. 31, pp. 72–81, 2011.

[4] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. Vol. 20, no. 1, pp. 46–61, 1973.

[5] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-Time Syst.*, vol. 28, no. 2-3, pp. 101–155, Nov. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:TIME.0000045315.61234.1e

[6] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Ph.D. dissertation, 1983.

[7] S. K. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, 1999.

[8] A. Srinivasan, "Efficient and flexible fair scheduling of real-time tasks on multiprocessors," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2003.

[9] R. Guerra, "A gravitational task model for target sensitive real-time applications," Ph.D. dissertation, Technische Universität Kaiserslautern, June 2011.

[10] S. K. Baruah, "Dynamic-and static-priority scheduling of recurring real-time tasks," *Real-Time Systems*, vol. 24, no. 1, pp. 93–128, 2003.

[11] ——, "The non-cyclic recurring real-time task model," in *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS)*, 2010, pp. 173–182.

[12] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes." in *Proceedings of the 33st IEEE Real-Time Systems Symposium (RTSS)*, 2012, pp. 63–72.

[13] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Information and Computation*, vol. 205, no. 8, pp. 1149–1172, 2007.

[14] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2011, pp. 71–80.

[15] ——, "On the tractability of digraph-based task models," in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2011, pp. 162–171.

[16] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed. Springer Science+Business Media, 2005.

[17] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed. Prentice Hall, 2001.

[18] ——, *Structured Computer Organization*, 5th ed. Prentice Hall, 2005.

[19] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[20] K. Jeffay, D. Stanat, and C. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, Dec 1991, pp. 129–139.

[21] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, 1971, pp. 151–158.

[22] M. Nasri, S. Baruah, G. Fohler, and M. Kargahi, "On the optimality of rm and edf for non-preemptive real-time harmonic tasks," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014, pp. 331:331–331:340. [Online]. Available: http://doi.acm.org/10.1145/2659787.2659806

[23] N. C. Audsley, *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Citeseer, 1991.

[24] E. Bini and G. Buttazzo, "A hyperbolic bound for the rate monotonic algorithm," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*. IEEE, 2001, pp. 59–66.

[25] G. Buttazzo, "Rate monotonic vs. edf: Judgment day," in *Embedded Software*, ser. Lecture Notes in Computer Science, R. Alur and I. Lee, Eds. Springer Berlin / Heidelberg, 2003, vol. 2855, pp. 67–83. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45212-6_6

[26] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. Vol. 2, no. 4, pp. 301–324, 1990.

[27] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of the 11th IEEE Real-Time Systems Symposium*. IEEE, 1990, pp. 182–190.

[28] M. L. Dertouzos, "Control robotics: The procedural control of physical processes," *Proceedings of the IFIP Congress*, pp. 807 – 813, 1974.

[29] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," in *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1993, pp. 345–354.

[30] H. Kopetz, "Event-triggered versus time-triggered real-time systems," in *Operating Systems of the 90s and Beyond*, ser. Lecture Notes in Computer Science, A. Karshmer and J. Nehmer, Eds. Springer Berlin Heidelberg, 1991, vol. 563, pp. 86–101. [Online]. Available: http://dx.doi.org/10.1007/BFb0024530

[31] A. Vrchoticky and P. Puschner, "On the feasibility of response time predictions — an experimental evaluation," in *PDCS Project (Esprit BRA Project 3092), Second Year Report*, 1991.

[32] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/1978802.1978814

[33] C. L. Liu, "Scheduling algorithms for multiprocessors in a hard real-time environment," *JPL Space Programs Summary*, vol. Vol. 37-60, pp. 28–31, 1969.

[34] J. E. G. Coffman, M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM Journal of Computing*, vol. 7(1), pp. 1–17, Feb. 1978.

[35] M. DiNatale and J. Stankovic, "Applicability of simulated annealing methods to real-time scheduling and jitter control," in *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, dec 1995, pp. 190 –199.

[36] G. Fohler, "Analyzing a pre run-time scheduling algorithm and precedence graphs," E182, Vienna, Austria, MARS 13/92, September 1992.

[37] S. K. Baruah, "Task partitioning upon heterogeneous multiprocessor platforms," in *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, may 2004, pp. 536 – 543.

[38] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. IEEE, 2001, pp. 193–202.

[39] B. Andersson and J. Jonsson, "Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition," in *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*. IEEE, 2000, pp. 337–346.

[40] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, dec. 2006, pp. 101 –110.

[41] S. K. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[42] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, 2006, pp. 322 –334.

[43] A. Gujarati, F. Cerqueira, and B. B. Brandenburg, "Multiprocessor real-time scheduling with arbitrary processor affinities: From practice to theory," *Real-Time Systems*, 2014.

[44] B. Nikolić, P. M. Yomsi, and S. M. Petters, "Worst-case communication delay analysis for many-cores using a limited migrative model," in *The 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2014.

[45] K. Hong and J.-T. Leung, "On-line scheduling of real-time tasks," in *Real-Time Systems Symposium, 1988., Proceedings.*, dec 1988, pp. 244 –250.

[46] M. Dertouzos and A. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *Software Engineering, IEEE Transactions on*, vol. 15, no. 12, pp. 1497 – 1506, dec 1989.

[47] N. Fisher, "The multiprocessor real-time scheduling of general task systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2007.

[48] J. S. JP Lehoczky, L Sha, "Enhanced aperiodic responsiveness in hard real-time environments," in *Proceedings of the 8th IEEE Real-Time Systems*, 1987.

[49] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, pp. 27–60, 1989, 10.1007/BF02341920. [Online]. Available: http://dx.doi.org/10.1007/BF02341920

[50] J. Strosnider, J. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, January 1995.

[51] M. Spuri and G. Buttazzo, "Efficient aperiodic service under earliest deadline scheduling," in *Proceedings of the 15th IEEE Real-Time Systems Symposium*, Pisa, Italy, 1994, pp. 2–11.

[52] ——, "Scheduling aperiodic tasks in dynamic priority systems," *Real-Time Systems*, vol. 10, pp. 179–210, 1996, 10.1007/BF00360340. [Online]. Available: http://dx.doi.org/10.1007/BF00360340

[53] G. C. Buttazzo and F. Sensini, "Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments," *IEEE Transactions on Computers*, vol. Vol. 48, no. 10, pp. 1035–1052, 1999.

[54] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, December 1998, pp. 4–13.

[55] J. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *Real-Time Systems Symposium*, 1992, pp. 110–123.

[56] R. Davis, K. Tindell, and A. Burns, "Scheduling slack time in fixed priority preemptive systems," in *Real-Time Systems Symposium, 1993., Proceedings.*, 1993.

[57] D. Isovic and G. Fohler, "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints," in *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, Orlando, Florida, USA, Nov. 2000, pp. 207–216.

[58] J. Luo and N. Jha, "Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems," in *Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on*, 2000, pp. 357–364.

[59] M. van den Heuvel, R. Bril, J. Lukkien, D. Isovic, and G. Ramachandran, "Rtos support for mixed time-triggered and event-triggered task sets," in *Intl. Conf. on Computational Science and Engineering (CSE)*, 2012.

[60] J. Theis and G. Fohler, "Mixed criticality scheduling in time-triggered legacy systems," in *1st Workshop on Mixed Criticality Systems, IEEE Real-Time Systems Symposium*, December 2013.

[61] S. Schorr and G. Fohler, "Online admission of non-preemptive aperiodic tasks in offline schedules," in *Proceedings of Work-in-Progress Session, 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, Brussels, Belgium, July 2010.

[62] ——, "Integrated time- and event-triggered scheduling - an overhead analysis on the arm architecture," in *The 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Taipei, Taiwan, August 2013.

[63] V. Nollet, D. Verkest, and H. Corporaal, "A safari through the mpsoc run-time management jungle," *Journal of Signal Processing Systems*, vol. 60, pp. 251–268, 2010, 10.1007/s11265-008-0305-4. [Online]. Available: http://dx.doi.org/10.1007/s11265-008-0305-4

[64] G. Gracioli and A. Frohlich, "An experimental evaluation of the cache partitioning impact on multicore real-time schedulers," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, Aug 2013, pp. 72–81.

[65] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 143–148.

[66] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-directed page coloring for multiprocessors," in *ACM SIGPLAN Notices*, vol. 31, no. 9. ACM, 1996, pp. 244–255.

[67] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*. IEEE, 2007, pp. 1–6.

[68] P. Burgio, M. Ruggiero, F. Esposito, M. Marinoni, G. Buttazzo, and L. Benini, "Adaptive tdma bus allocation and elastic scheduling: A unified approach for enhancing robustness in multi-core rt systems," in *Computer Design (ICCD), 2010 IEEE International Conference on*, oct. 2010, pp. 187 –194.

[69] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 251–256.

[70] M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, 2009.

[71] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the multi-processor SoC design space with SystemC," *The Journal of VLSI Signal Processing*, vol. 41, pp. 169–182, 2005. [Online]. Available: http://dx.doi.org/10.1007/s11265-005-6648-1

[72] "Adaptivity and control of resources in embedded systems - ACTORS," http://www.actors-project.eu/, accessed: 2014-11-28.

[73] H. Kopetz, "Sparse time versus dense time in distributed real-time systems," in *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992, pp. 460–467.

[74] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 933–940, aug. 1987.

[75] G. Fohler, "Flexibility in statically scheduled real-time systems," Ph.D. dissertation, Technische Universität Wien, Wien, Austria, April 1994.

[76] M. Dales, "SWARM - ARM7 processor simulator," http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html, 2014.

[77] C.-J. Hou and K. G. Shin, "Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1338–1356, 1997.

[78] P. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *System Architecture*, 2013.

[79] M. Stephen, *Machine Learning: An Algorithmic Perspective*. CRC Press, 2009.

[80] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, 2005.

[81] S. Schorr and G. Fohler, "Simulation results for slot shifting, response times and acceptance ratios," Technische Universität Kaiserslautern, Germany, Tech. Rep., March 2013.

[82] H. A. Sturges, "The choice of a class interval," *Journal of the American Statistical Association*, vol. 21, no. 153, pp. 65–66, 1926.

[83] W. Sun and Y.-x. Yuan, *Optimization Theory and Methods*. springer, 2006, vol. 1.

[84] K. J. Nesbit, J. E. Smith, M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero, "Multicore resource management," *IEEE micro*, vol. 28, no. 3, pp. 6–16, 2008.

[85] A. K. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*. IEEE, 2001, pp. 75–84.

[86] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Transactions on Networking (ToN)*, vol. 6, no. 5, pp. 611–624, 1998.

[87] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk scheduling with quality of service guarantees," in *Multimedia Computing and Systems, 1999. IEEE International Conference on*, vol. 2. IEEE, 1999, pp. 400–405.

[88] "D-Bus, an open-source inter-process communication protocol," http://www.freedesktop.org/wiki/Software/dbus, accessed: 2014-11-28.

[89] "The GNU Linear Programming Kit (GLPK)," https://www.gnu.org/software/glpk/, accessed: 2014-11-28.

[90] A. Kotra and G. Fohler, "Demo : Resource aware real-time stream adaptation for mpeg-2 transport streams in constrained bandwidth networks," in *Proceedings of IEEE International Conference on Multimedia and Expo (ICME)*, singapore, July 2010.

[91] J. Theis, G. Fohler, and S. Baruah, "Schedule table generation for time-triggered mixed criticality systems," in *1st Workshop on Mixed Criticality Systems, IEEE Real-Time Systems Symposium*, December 2013.

[92] "Litmus RT - linux testbed for multiprocessor scheduling in real-time systems," http://www.cs.unc.edu/~anderson/litmus-rt/, accessed: 2014-11-28.

Summary

The efficient exploitation of real-time systems based on multicore platforms creates new challenges: First, there is the need for effective *scheduling algorithms*. On the one hand, these algorithms must feature low overheads to improve the use of the computational resources of real-time systems. On the other hand, these algorithms must ensure timely execution of tasks, i.e., provide runtime guarantees for deadlines constrained applications. Furthermore, the algorithms must provide sufficient flexibly react to unforeseen events.
Second, multicore systems concurrently run multiple applications which compete for the scarce resources of the system. This leads to contention for shared hardware resources and complicates system analysis. Without an adaptive *resource management*, availability of resources cannot be guaranteed to the real-time applications. It is a challenging task to implement an effective and efficient resource management which is capable to handle all these applications with their varying resource requirements. The main goal of resource management is to guarantee a minimum resource availability to real-time applications. A further goal is to fulfill global optimization objectives, e.g., maximization of the global system performance, or the user perceived quality of service.

In this thesis, we derived methods based on the *slot shifting* algorithm [1]. Slot shifting flexibly schedules time-constrained applications, ensure timely execution of tasks, and can react to unforeseen events in time-triggered systems. For this reason, we aim at designing slot-shifting-based algorithms targeted for multicore systems to tackle the aforementioned challenges.

The main contribution of this thesis is to present two global multicore slot shifting algorithms. Furthermore, we extensively analyzed and improved the different slot shifting algorithms in various ways. Finally, we implemented a slot-shifting-based logic into a resource management framework. Therefore, we proved that the slot shifting algorithm effectively and efficiently performs adaptive resource management on multicore systems.

The following sections summarize the contents of each individual chapter of this thesis in more details.

Chapter 1

This chapter contains the introduction to the thesis. First, this chapter started with a brief description of the problems that this thesis dealed with in section 1.1. After

that, section 1.2 introduced our approach to tackle these problems. It also listed the contributions of this thesis to the state-of-the-art of real-time scheduling theory. Next, section 1.3 explained basic terms and principles and presented the related work in this area. Finally, section 1.4 detailed the outline of the rest of the thesis.

Chapter 2

This chapter presented the original slot shifting algorithm [1], which consists of two main phases: the offline phase and the online phase. In the first part of the chapter, we explained the concepts underlying the offline phase. The offline phase in a first step resolves the complex constraints of offline tasks using an offline scheduler, which creates an offline scheduling table. Then, the next step is to annotate this offline table with additional information about spare capacities, i.e., task flexibilities.

In the second part of this chapter, we described the working-principle of the online phase: The online phase uses a runtime scheduler, which at runtime makes twofold use of the annotated offline scheduling table: First, it dispatches the jobs of the tasks according to the offline table in order to obey the offline tasks' complex constraints. If multiple jobs are ready to be scheduled simultaneously, dispatching of jobs happens according to EDF. The second use of the offline scheduling table is to facilitate feasible integration of jobs of aperiodic tasks at runtime. Every aperiodic job arrival triggers an online acceptance test, which in case of success triggers a guarantee algorithm to update the schedule and thus integrate the aperiodic job.

Chapter 3

This chapter, entitled "*Online Admission of Non-Preemptive Aperiodic Tasks in Time-Triggered Schedules*", illustrated our approach to include the handling of non-preemptive firm aperiodic tasks into time-triggered systems based on the slot shifting algorithm [61]. The challenge hereby was to determine whether or not the firm non-preemptive aperiodic job can be guaranteed based on the particular offline table. In this chapter, we proposed a new methodology for a modified acceptance test. The underlying idea of our approach is to cycle through the intervals, starting in the current interval up to the interval hosting the deadline of the non-preemptive job. The acceptance test identifies jobs that can be shifted to maximize the consecutive slots for non-preemptive execution. If sufficient available spare capacity in consecutive slots is found, the guarantee algorithm is invoked for the non-preemptive job. However, when accepting one or multiple non-preemptive jobs, there is a risk of a resulting decreased overall acceptance ratio for aperiodic jobs. Thus, we proposed different approaches to guarantee the non-preemptive task to allow for tuning the algorithm towards minimized response time of the non-preemptive job or flexibility for future aperiodic job arrivals. This trade-off needs to be decided depending on the specific requirements and constraints of the system to be designed.

The main advantages of the approach are that our new acceptance test induces only minimal memory overhead and features a runtime complexity of $O(n^2)$. Additionally, our approach does not require creating a new additional offline scheduling table, instead it updates an existing table.

Chapter 4

This chapter, named "*Event-Triggered Activities in Time-Triggered Schedules on Multi-core Systems*", presents our global slot shifting algorithms targeted for multicore systems. We examined the challenges associated with the design of global slot shifting algorithms. In previous work, other researchers have proven the non-existence of an optimal multiprocessor online scheduling algorithm for aperiodic tasks. For this reason, we aimed to find effective heuristic-based algorithms to globally schedule aperiodic tasks with low runtime overheads.

This chapter discussed the issues of different methods to tackle the problem: One issue arises on the decision how to store the offline table data. While global storage potentially requires less memory and is thus more efficient, it suffers from delays caused by contention due to concurrent accesses by multiple cores. Storing the table data on the individual cores, i.e., distributed storage, is more demanding in terms of resources. When the table is stored on the individual cores, every core needs to update its data and potentially more memory is required in total. Nevertheless, distributed storage is not subject to overheads caused by contention.
Another challenge arises when aperiodic jobs cannot be guaranteed locally: how to quickly determine a suitable core for the aperiodic job? A further challenge is to reduce unnecessary synchronization overheads due to which cores slow down each other. This challenge is closely related to the need to reduce the data exchange among the cores to a minimum.

After discussing the challenges, the chapter presented our two global scheduling algorithms: a spare-capacity-based and a negotiation-based version of the slot shifting algorithm. In the first algorithm, every core "announces" its available spare capacity to the other cores via shared memory. When a core needs to delegate an aperiodic job, the available spare capacity of the other cores is used as a metric to decide to which core to send the job to. In other words, this heuristic-based algorithm aims at increasing the probability of successful acceptance of the aperiodic job at the destination core.
The second algorithm also aims at maximizing this probability. In order to achieve this, it triggers all remaining cores to perform an acceptance test for the locally rejected job. The first core which successfully tests the aperiodic job integrates it locally, while the others continue with the next aperiodic jobs—if there are any. The advantage of the latter algorithm is that it relieves the delegating core from the computationally intense process of determining a suitable destination core. Instead, the remaining cores of the system negotiate among themselves a suitable candidate.

Chapter 5

In this chapter, entitled "*Efficiency Evaluation*", we described a variety of experiments to analyze the different slot shifting algorithms in terms of efficiency; evaluation in terms of effectiveness is performed in the succeeding chapter. Using MPARM, a cycle-accurate MP-SoC simulator which was developed at the MicRel Lab at the University of Bologna [71], we determined the runtime overheads that are associated with the slot shifting algorithms.

The experiments in this chapter quantified the efficiency of the algorithms: The simulations were performed on a simulated ARM multicore system and used a "bare-board" C-implementation of the respective slot shifting algorithms. Since there exists no disturbing operating systems and since other interferences do not exist, the results of the experiment are reproducible. Based on this runtime analysis, we developed and analyzed another approach, called SDL, to reduce the runtime overheads, which yielded an improved guarantee algorithm.

Chapter 6

While the previous chapter evaluated the efficiency of the slot shifting algorithms, this chapter described the experiments that we conducted to evaluate the effectiveness of the slot shifting algorithm. In order to achieve this, we ran the experiments with a vast number of randomly created task sets on Elwetritsch, the high performance cluster of the University of Kaiserslautern. We scheduled the same task sets using the different slot shifting algorithms. Additionally, we also applied EDF with background processing of the aperiodic jobs as a comparison metric. We determined the maximum achievable acceptance ratio within 95% confidence intervals, analyzed the average number of performed acceptance tests and measured the resulting response times of aperiodic jobs. The experiments showed that using SDL is always beneficial, as it significantly improves the response times of the aperiodic jobs for all slot shifting algorithms. This improvement comes at zero cost, since the acceptance ratio remains virtually constant. The results showed that the partitioned slot shifting always outperforms EDF with background processing of the aperiodic jobs. As expected, they also proved that our global slot shifting algorithms always outperform the original, partitioned algorithm.

Chapter 7

This chapter addressed the resource management on multicore platforms using slot-shifting-based algorithms. It presented our approach to implement a slot-shifting-based logic into a generic adaptive resource management framework. The system design is based on the system design of the ACTORS project [3, 72]. The ACTORS framework runs on top of a patched Linux kernel. This kernel features real-time capabilities and employs a constant bandwidth server that ensures temporal isolation between the different concurrently running applications. In the heart of the framework, there sits the resource manager, a privileged user space application which controls and manages the system resources among the running applications.

Our implementation serves as a proof of concept: It confirms the applicability of the slot shifting algorithm to multicore resource management. As a result, we showed the validity of our approach and we could bridge the gap between real-time scheduling theory and real-world implementations.

Chapter 8

The last chapter drew the conclusions from the results of all the experiments and summarized the main contributions of this thesis. It gave pointers for future directions of research and concluded the thesis with final remarks.

Appendices

In the different appendices attached to the main body of this thesis, further details and additional figures can be found that were not placed in the respective chapters: Appendix A listed more detailed results of MPARM Experiment 4, i.e., runtime overheads for different categories of slots, obtained on a simulated quadcore system. Similarly, appendix B listed the detailed results of MPARM Experiments 5a and b which analyzed the impact of the number of jobs in the job set on the runtime of the slot shifting algorithms. Finally, the appendices C–G contained additional results of the Linux Experiments 1–5.

Zusammenfassung
Adaptives Echtzeit-Scheduling und Ressourcen-Management auf Mehrkernprozessor-Architekturen

Echtzeitsysteme sind Rechensysteme, die einerseits Daten verarbeiten und Ergebnisse berechnen müssen, und andererseits Fristen hinsichtlich der Dauer dieser Berechnungen einhalten müssen. Das bedeutet, Echtzeitsysteme müssen sowohl die *Korrektheit*, als auch die *Rechtzeitigkeit* der Ergebnisse garantieren. Um diese strikten Fristen einzuhalten, nutzen Echtzeitsysteme *Echtzeit-Scheduling-Algorithmen*. Ein Scheduling-Algorithmus bestimmt die Ausführungsreihenfolge der Jobs auf den Prozessoren des Systems. Heute werden Echtzeitsysteme überall in der Industrie eingesetzt, nicht nur in sicherheitskritischen Systemen, sondern zum Beispiel auch in Kommunikations-, Unterhaltungs- und Multimedia-Systemen.

Mit dem Aufkommen von Mehrkernprozessor-Plattformen haben sich neue Herausforderungen für die effiziente Nutzung von Echtzeitsystemen ergeben: Erstens die Notwendigkeit für effektive Scheduling-Algorithmen mit niedrigen Overheads, um die Nutzung der Rechenkapazitäten des Echzeitsystems zu optimieren. Zusätzlich benötigen viele Systeme einen Scheduling-Algorithmus, der flexibel auf unerwartete Ereignisse reagiert.
Zweitens führt der den Mehrkernprozessor-Systemen inhärente Parallelismus zu einem Wettstreit um die geteilten Hardware-Ressourcen und kompliziert dadurch die Systemanalyse. Zu jedem beliebigen Zeitpunkt laufen mehrere Applikationen mit veränderlichen Ressourcenanforderungen und wetteifern um die knappen Systemressourcen. Daraus folgt unweigerlich das Bedürfnis nach einem adaptiven *Ressourcen-Management*. Dabei stellt die Implementierung und das Erzielen eines effektiven und effizienten Ressourcen-Managements keine triviale Herausforderung dar. Das Hauptziel des Ressourcen-Managements ist es eine minimale Ressourcenverfügbarkeit für die laufenden Echtzeit-Applikationen zu garantieren. Ein weiteres Anliegen ist es, globale Zielvorstellungen durchzusetzen, wie zum Beispiel die Maximierung der globalen Systemleistung, oder die Maximierung der vom Benutzer wahrgenommenen Servicequalität des Systems.

In dieser Dissertation werden Methoden und Algorithmen entwickelt und präsentiert, die auf dem Slot-Shifting-Algorithmus [1] basieren. Eine interessante Eigenschaft dieses Algorithmus ist es, dass er die Vorteile von *zeitgesteuerten* und *ereignisgesteuerten* Scheduling-Algorithmen zu vereinen sucht: Slot-Shifting ist ein Algorithmus für das Scheduling von Applikationen, die Fristen beachten müssen. Darüber hinaus ist er in der Lage auf unerwartete Ereignisse in zeitgesteuerten Systemen flexibel zu reagieren. Aus diesem Grund stellt dieser Algorithmus einen geeigneten Ausgangspunkt für den Entwurf von Algorithmen für Mehrkernprozessor-Systeme zur Meisterung der vorher erwähnten Herausforderungen dar.

Der Hauptbeitrag dieser Dissertation ist der Entwurf zweier neuer globaler Slot-Shifting-Algorithmen für Mehrkernprozessor-Systeme. Zusätzlich erweitert diese Dissertation den Originalalgorithmus mit der Zielsetzung sein Laufzeitverhalten zu verbessern, oder um nicht-unterbrechbare aperiodische Tasks mit Fristen zu unterstützen. Die Effektivität und Effizienz der resultierenden Algorithmen wird mittels einer Vielzahl von Experimenten evaluiert und bestätigt.

Im vorletzten Kapitel wird eine Implementierung eines Slot-Shifting-basierenden Logikmoduls für ein Ressource-Management-Framework für Mehrkernprozessor-Systeme vorgestellt. Damit gelingt es eine Brücke zu schlagen zwischen Echtzeit-Scheduling-Theorie auf der einen Seite und einer praxisnahen Implementierung auf der anderen Seite. Die Ergebnisse der Implementierung beweisen die Anwendbarkeit des Slot-Shifting-Algorithmus für effektives und effizientes adaptives Ressourcen-Management auf Mehrkernprozessor-Systemen.

Kapitel 1

Dieses Kapitel beinhaltet die Einleitung der Dissertation und beginnt in Abschnitt 1.1 mit einer kurzen Beschreibung der Probleme, die diese Dissertation behandelt. Danach folgt Abschnitt 1.2, der die Methode beschreibt, mit der diese Probleme behandelt werden. Außerdem listet dieser Abschnitt auch die Beiträge dieser Dissertation zum State-of-the-Art der Echtzeit-Scheduling-Theorie auf. Der nächste Abschnitt erklärt grundlegende Begriffe und Prinzipien und präsentiert bereits vorliegende themenverwandte Arbeiten anderer Forscher auf diesem Gebiet. Danach folgt Abschnitt 1.4, der einen Überblick über den Inhalt der restlichen Kapitel dieser Dissertation gibt.

Kapitel 2

Dieses Kapitel präsentiert den *Slot-Shifting-Algorithmus* [1], der aus zwei Hauptphasen besteht: der *Offline Phase* und der *Online Phase*. Im ersten Teil des Kapitels werden die der Offline Phase zugrunde liegenden Konzepte erklärt. Die Offline Phase nimmt sich in einem ersten Schritt der komplexen Bedingungen und Einschränkungen eines gegebenen Tasksets an. Dazu bedient sie sich eines Offline Schedulers, der eine *Offline Scheduling Tabelle* erstellt. Im nächsten Schritt annotiert die Offline Phase diese Tabelle mit zusätzlichen Informationen über die Flexibilität der einzelnen Tasks im Schedule, den sogenannten *Spare Capacities*.

Im zweiten Teil dieses Kapitels wird die Arbeitsweise der Online Phase beschrieben: Die Online Phase nutzt einen Scheduler, der zur Laufzeit des Systems die annotierte Offline Scheduling Tabelle auf zweierlei Art und Weise nutzt: Erstens wählt er die Jobs der Tasks unter Berücksichtigung der Offline Tabelle, damit die komplexen Bedingungen und Einschränkungen der Tasks eingehalten werden. Falls mehrere Jobs gleichzeitig ausführungsbereit sind, wählt der Scheduler den Job mit der frühesten Frist zuerst, d.h., er wendet den Earliest-Deadline-First (EDF) Algorithmus an. Zweitens erlaubt es die Offline Scheduling Tabelle, Jobs von aperiodischen Tasks zur Laufzeit so zu integrieren, dass andere Jobs von anderen Tasks davon nicht gestört werden. Jedes Mal wenn ein aperiodischer Job startet, triggert Slot-Shifting einen Akzeptanz-Test. Wenn der Test erfolgreich verläuft, wird nachfolgend ein Garantie-Algorithms auf diesen aperiodischen Job angewandt, welcher den aperiodischen Job in das Schedule integriert. Andernfalls wird der aperiodische Job verworfen, da nicht genügend Rechenkapazität garantiert werden kann, um ihn sicher bis Ablauf seiner Frist komplett auszuführen.

Kapitel 3

Das Kapitel *"Online Admission of Non-Preemptive Aperiodic Tasks in Time-Triggered Schedules"* illustriert eine neue Methode Jobs nicht-unterbrechbarer aperiodischer Tasks, die Fristen beachten müssen, mittels dem Slot-Shifting-Algorithmus in zeitgesteuerte Systeme zu integrieren [61]. Die Herausforderung hierbei ist es, herauszufinden ob diese Jobs, basierend auf einer gegebenen Offline Scheduling Tabelle, garantiert werden können. Das Kapitel präsentiert eine neue Methode, die auf einem modifizierten Akzeptanz-Test basiert.

Die zugrunde liegende Idee dieser Methode ist folgende: Durch die annotierte Offline Tabelle sind Zeit-Intervalle vorgegeben. Die Methode iteriert über diese Intervalle, von dem momentan aktuellen Intervall bis zu dem Intervall welches die Frist des nicht-unterbrechbaren aperiodischen Jobs beinhaltet. Dabei identifiziert der Akzeptanz-Test diejenigen Jobs, die das Potenzial haben die Anzahl freier nebeneinanderliegender Slots zu maximieren. Anders ausgedrückt, es werden diejenigen Jobs identifiziert, durch deren Verschiebung im Schedule mehr Slots für die Ausführung des nicht-unterbrechbaren Jobs zur Verfügung gestellt werden können. Falls genügend solcher nebeneinanderliegender Slots gefunden werden, wird ein Garantie-Algorithmus aufgerufen, der den nicht-unterbrechbaren Job in das Schedule integriert. Andernfalls wird der Job verworfen, da seine fristgerechte Ausführung nicht garantiert werden kann.

Die Integration eines oder mehrerer nicht-unterbrechbarer Jobs führt zu einem immanenten Risiko einer resultierenden verringerten Akzeptanzrate für aperiodische Jobs. Um dem entgegen zu wirken, diskutiert das Kapitel die Möglichkeit, den Garantie-Algorithmus zu modifizieren: Eine Möglichkeit ist es, den Algorithmus so zu ändern, dass die Antwortzeit (Response Time) der Jobs minimiert wird. Die andere Möglichkeit ist es den Algorithmus so zu modifizieren, dass er mehr Flexibilität für zukünftig startende aperiodische Jobs ermöglicht. Dieser Zielkonflikt muss jeweils entsprechend den spezifischen Anforderungen und Beschränkungen des zu designenden Systems gelöst werden.

Die Hauptvorteile dieser neuen Methode sind, dass der neue Akzeptanz-Test nur einen minimalen Speicheroverhead erzeugt, und dass die Methode eine Laufzeitkomplexität von $O(n^2)$ hat. Des Weiteren braucht diese Methode keine neue Offline Scheduling Tabelle zu berechnen, sondern aktualisiert einfach die bereits existierende.

Kapitel 4

Das Kapitel *"Event-Triggered Activities in Time-Triggered Schedules on Multicore Systems"* präsentiert neue globale Slot-Shifting-Algorithmen für Mehrkernprozessor-Systeme. Es behandelt die Herausforderungen die mit dem Design von globalen Slot-Shifting-Algorithmen verbunden sind. Die Ergebnisse anderer Forscher belegen die Nicht-Existenz von optimalen online Mehrkernprozessor-Scheduling-Algorithmen für aperiodische Tasks. Aus diesem Grund ist die Zielsetzung dieser Arbeit effektive Heuristik-basierte Algorithmen zu finden, die in der Lage sind, aperiodische Tasks mit geringen Overheads zu schedulen.

Dieses Kapitel diskutiert verschiedene Ansätze dieses Problem zu lösen und analysiert die sich daraus ergebenden Vor- und Nachteile. Ein Problem involviert die Frage wie die Daten aus der Offline Tabelle am besten gespeichert werden sollten: global oder verteilt? Während es möglich ist die Daten global effizienter zu speichern, d.h. mit weniger Speicherplatzbedarf, schafft die globale Speicherung der Daten neue Probleme: Wenn mehrere Kerne gleichzeitig auf die Daten zugreifen wollen, entstehen Zugriffskonflikte und dadurch Overheads. Die verteilte Speicherung, d.h. die individuelle Speicherung der Daten auf den einzelnen Kernen, stellt höhere Anforderungen an die Systemressourcen: Jeder einzelne Kern muss seine individuellen Daten selbst managen und dafür sorgen diese aktuell zu halten. Dies kann im Gesamten zu mehr Speicherplatzverbrauch führen. Der Hauptvorteil der verteilten Speicherung der Daten ist, dass dadurch Overheads durch Zugriffskonflikte verringert oder ganz vermieden werden können.
Ein weiteres Problem entsteht, wenn aperiodische Jobs nicht lokal integriert werden können: Wie kann schnell ein geeigneter Kern für den aperiodischen Job identifiziert werden? Wiederum ein anderes Problem ist es unnötige Synchronisations-Overheads, durch die ein Kern den anderen verzögert, zu vermeiden oder zumindest zu verringern. Eine ähnliches Problem stellt die Minimierung des Datenaustausches zwischen den Kernen dar.

Nach der Diskussion all dieser Herausforderungen werden in diesem Kapitel zwei neue globale Scheduling-Algorithmen präsentiert: eine Spare-Capacity-basierte und eine Verhandlungs-basierte Version des Slot-Shifting-Algorithmus. Im ersten Algorithmus "verkündet" jeder Kern allen anderen Kernen im System mittels geteilten Speichers (shared memory) seine verfügbare Spare Capacity. Wenn ein Kern einen aperiodischen Job an einen anderen Kern delegieren muss, dient die verfügbare Spare Capacity der anderen Kerne als Entscheidungsfindungsmetrik für einen geeigneten Kern. Anders ausgedrückt, dieser Heuristik-basierte Algorithmus zielt darauf ab, die Wahrscheinlichkeit einer erfolgreichen Integration des aperiodischen Jobs auf dem "Zielkern" zu erhöhen.
Die Zielsetzung des zweiten Algorithmus ist es ebenfalls diese Wahrscheinlichkeit zu erhöhen. Dazu triggert der Algorithmus alle anderen Kerne im System, damit diese im nächsten Slot einen Akzeptanz-Test für diesen lokal abgelehnten aperiodischen Job

durchführen. Der erste Kern, der diesen Job erfolgreich testet, integriert ihn, während die anderen Kerne damit fortfahren, weitere eventuell vorhandene aperiodische Jobs zu testen.

Der Vorteil des zweiten Algorithmus ist es, dass er den versendenden Kern von der rechenintensiven Aufgabe befreit einen geeigneten Kern für den aperiodischen Job finden zu müssen. Stattdessen handeln die verbleibenden Kerne im System untereinander einen geeigneten Kandidaten aus.

Kapitel 5

Das Kapitel *"Efficiency Evaluation"* beschreibt eine Vielzahl von Experimenten, welche die Effizienz der verschiedenen Slot-Shifting-Algorithmen eruieren; ihre Effektivität wird im anschließenden Kapitel 6 untersucht. Allen Experimenten in diesem Kapitel gemein ist, dass sie auf dem MPARM, einem Zyklus-genauen Multiprozessor System-On-A-Chip (MP-SoC) Simulator ausgeführt werden. Der MPARM wurde im MicRel Lab an der Universität von Bologna entwickelt [71]. Dieser Simulator ist in der Lage, ein aus mehreren ARM Kernen bestehendes Mehrkernprozessor-System komplett mit Level-1-Caches und dazugehörigem Hauptspeicher zu simulieren.

Mittels MPARM wurden die Overheads der verschiedenen Slot-Shifting-Algorithmen zur Laufzeit bestimmt. Dazu wurde eine Implementierung der Algorithmen in der Hochsprache C in die verschiedenen simulierten Kerne des MPARM geladen. Dadurch dass es keine Störungen durch ein Betriebssystem oder durch andere Quellen gibt, sind alle Ergebnisse reproduzierbar. Des Weiteren wurde, basierend auf dieser Laufzeitanalyse, ein anderer Ansatz entwickelt und analysiert die Overheads zu reduzieren. Dieser Ansatz, genannt *SDL*, mündete in verbesserten Slot-Shifting-Algorithmen mit modifiziertem, d.h. schnellerem Garantie-Algorithmus.

Kapitel 6

Während das vorhergegangene Kapitel die Effizienz der Slot-Shifting-Algorithmen evaluiert hat, beschreibt dieses Kapitel die Experimente zur Evaluation der Effektivität der Slot-Shifting-Algorithmen. Daher wurden verschiedene Experimente mit einer großen Anzahl von zufällig erzeugten Tasksets auf der Elwetritsch, dem Hochleistungsrechner der Universität Kaiserslautern, durchgeführt. Dabei wurde jedes einzelne Taskset mit den verschiedenen Slot-Shifting-Algorithmen gescheduled. Als zusätzlicher Algorithmus wurde EDF mit Ausführung der aperiodischen Jobs im Hintergrund implementiert, um eine weitere Vergleichsmetrik zur Beurteilung zur Verfügung zu haben.

Bei diesen Experimenten wurde die maximal erreichbare durchschnittliche Akzeptanzrate für aperiodische Jobs (mit 95-prozentigen Konfidenzintervallen) bestimmt. Außerdem wurden die durchschnittliche Anzahl der durchgeführten Akzeptanz-Tests pro aperiodischen Job sowie die resultierenden Antwortzeiten (Response Times) der aperiodischen Jobs analysiert.

Die Ergebnisse der Experimente belegen, dass SDL immer Vorteile bringt, da der verbesserte Garantie-Algorithmus die Antwortzeiten der aperiodischen Jobs für alle Slot-Shifting-Algorithmen verbessert. Interessanterweise verursacht SDL dabei keine mess-

baren Nachteile, d.h., die Akzeptanzrate für aperiodische Jobs bleibt konstant. Des Weiteren zeigen die Ergebnisse, dass der original Slot-Shifting-Algorithmus immer bessere Ergebnisse liefert als EDF, welches die aperiodischen Jobs im Hintergrund ausführt. Wie erwartet, liefern die beiden globalen Slot-Shifting-Algorithmen immer deutlich bessere Ergebnisse als der Originalalgorithmus.

Kapitel 7

In diesem Kapitel geht es um das Problem des Ressourcen-Managements auf Mehrkernprozessor-Plattformen. Es beschreibt eine Implementierung einer auf Slot-Shifting basierenden Logik in einem generischen, adaptiven Resource-Management-Framework. Das Design des Systems basiert auf dem Systemdesign des ACTORS Projekts [3, 72]. Das ACTORS Framework läuft auf einem speziell gepatchten Linux Kernel, der sich durch seine Echtzeitfähigkeiten auszeichnet. Dieser Kernel nutzt eine Implementierung eines Constant-Bandwidth-Servers (CBS), was es ermöglicht, eine zeitliche Isolierung zwischen den einzelnen, gleichzeitig laufenden Applikationen zu etablieren. Im Herzen des Frameworks residiert der Resource-Manager, eine in C++ implementierte Applikation, ausgestattet mit administrativen Zugriffsrechten. Der Resource-Manager kontrolliert und managt die den laufenden Applikationen zugeordneten Systemressourcen.

Die Implementierung dieser auf Slot-Shifting basierenden Logik für den Resource-Manager dient als Machbarkeitsbeweis: Sie bestätigt die Anwendbarkeit des Slot-Shifting-Algorithmus auf das Mehrkernprozessor-Ressourcen-Management-Problem. Dadurch konnte eine Brücke geschlagen werden zwischen Echtzeit-Scheduling-Theorie auf der einen Seite und einer praxisnahen Implementierung auf der anderen Seite.

Kapitel 8

Das letzte Kapitel zieht Schlussfolgerungen basierend auf den Ergebnissen und Erkenntnissen der Experimente und fasst die Hauptbeiträge dieser Dissertation zusammen. Außerdem gibt es Hinweise für mögliche zukünftige Forschungen und beendet die Dissertation mit den Schlussbemerkungen.

Appendices

In den verschiedenen Appendices finden sich weitere detaillierte Ergebnisse und Grafiken, die aus Platz- und Übersichtlichkeitsgründen nicht im Hauptteil der Dissertation aufgeführt sind. In den jeweiligen Kapiteln der Dissertation sind entsprechende Hinweise platziert.
Appendix A listet weitere detaillierte Ergebnisse von MPARM Experiment 4: Die auf einem simulierten Quadcore-System gemessenen Overheads für die verschiedenen Kategorien von Slots. Analog listet Appendix B die detaillierten Ergebnisse der MPARM Experimente 5a und b, die den Einfluß der Jobsetgröße auf die Laufzeiten der verschiedenen Slot-Shifting-Algorithmen analysieren. Schließlich beinhalten die Appendices C–G zusätzliche Ergebnisse der Linux Experimente 1–5.

CURRICULUM VITAE

Stefan Schorr

PERSONAL DATA

Last Name:	Schorr
First Name:	Stefan
Nationality:	German
Office Address:	Chair of Real Time Systems, TU Kaiserslautern, Postfach 3049, 67653 Kaiserslautern, Germany

EDUCATION

Since October 2009	Ph.D. candidate at the Department of Real-Time Systems at the University of Kaiserslautern
September 2009	Diplom-Ingenieur (Dipl.-Ing.) Informationstechnik (equivalent to Master of Science in Information Technology) from the University of Kaiserslautern, Germany
April 2008 – November 2008	Student Research Project at the Michigan State University (MSU), East Lansing, USA
October 2004	Undergraduate studies in Information Technology at the University of Kaiserslautern, Germany

RESEARCH INTERESTS

Since October 2009, I have been working in the department of Real-Time Systems at the University of Kaiserslautern, Germany. The department is a core partner of the ARTIST Network of Excellence on Embedded System Design.

My research interests include multicore systems, real-time scheduling theory and adaptive and flexible real-time systems. The focus is to improve the efficiency of multicore embedded systems to avoid waste of energy and the integration of event-triggered activities into time-triggered systems.

From October 2009 until January 2011, I participated at the ACTORS (Adaptivity and Control of Resources in Embedded Systems) project led by Ericsson AB. ACTORS is an EU/IST FP7 STREP on feedback-based resource management and data-flow modeling in embedded systems. I was involved with the implementation of the ACTORS Resource Manager and its testing and integration into the ACTORS framework as well as with the integration of an adaptive MPEG-4 frame skipper application. Furthermore, I developed a logic module based on the slot shifting algorithm for the Resource Manager.

TEACHING EXPERIENCE

In the winter term 2009, I supervised the laboratory of the course Real-Time Systems II. Starting from the summer term 2010, I have been in charge of the exercise of the course Real-Time Systems I. And since winter term 2012, I have been responsible for the Real-Time Systems laboratory.

PUBLICATIONS

JOURNAL PAPERS

Enrico Bini, Giorgio Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Karl-Erik Årzén, Vanessa Romero Segovia and Claudio Scordino, *Resource Management on Multi-core Systems: the ACTORS approach*, IEEEMicro, May 2011.

REFEREED CONFERENCE AND WORKSHOP PAPERS

Stefan Schorr and Fohler Gerhard, *Integrated Time- and Event-triggered Scheduling - An Overhead Analysis on the ARM Architecture,* Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2013), Taipei, Taiwan, August 2013.

Stefan Schorr, Anand Kotra, Gerhard Fohler, Johan Eker, Karl-Erik Årzén and Vanessa Romero Segovia, *Adaptive Resource Management in the ACTORS Framework - A Live DVB-T/Webcam Demo*, RTSS@Work Demo Session of the 32nd IEEE Real-Time Systems Symposium, November 2011

Karl-Erik Årzén, Vanessa Romero Segovia, Stefan Schorr and Gerhard Fohler, *Reservation-Based CPU Management for Multicore Platforms*, RTiS2011: Real-Time in Sweden, Mälardalen University, Västerås, Sweden, June 2011.

Karl-Erik Årzén, Vanessa Romero Segovia, Stefan Schorr and Gerhard Fohler, *Adaptive Resource Management Made Real*, Proceedings of the 3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2011), Chicago, USA, April 2011.

Karl-Erik Årzén, Vanessa Romero Segovia, Mikael Kralmark, Stefan Schorr, Anand Kotra and Gerhard Fohler, *Demo: Adaptive Resource Management Made Real*, Proceedings of the 3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2011), Chicago, USA, April 2011.

Stefan Schorr and Gerhard Fohler, *Online Admission of Non-Preemptive Aperiodic Tasks in Offline Schedules*, Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010), Brussels, Belgium, Work in Progress Session, July 2010.

Vanessa Romero Segovia, Karl-Erik Årzén, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Johan Eker and Harald Gustafsson, *Adaptive Resource Management for Mobile Terminals - The ACTORS approach*, Proceedings of 1st Workshop on Adaptive Resource Management (WARM2010), Stockholm, Sweden, April 2010.